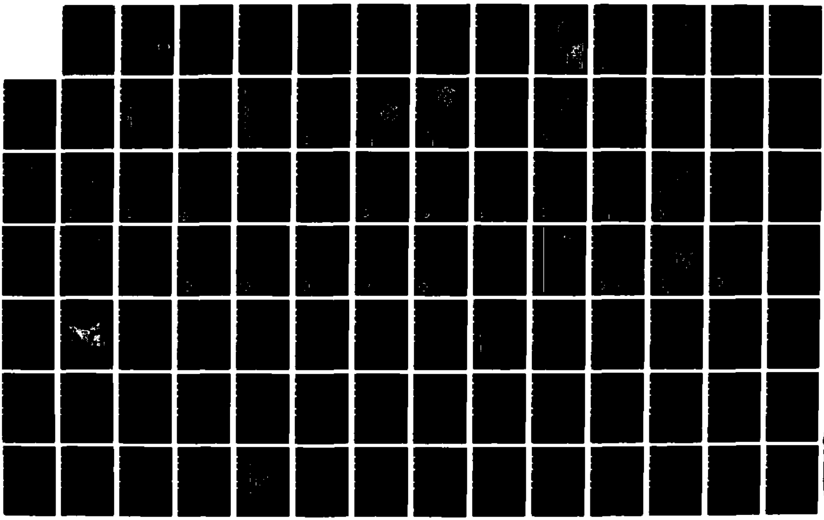
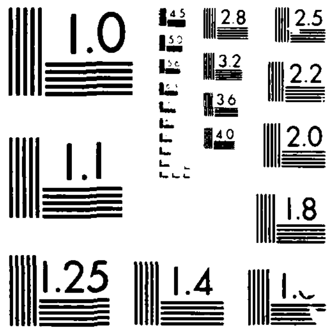
AD-A192 074 ASEET ADVANCED ADA WORKSHOP JANUARY 1988(U) ADA JOINT PROGRAM OFFICE ARLINGTON VA JAN 88 1/5

UNCLASSIFIED F/G 12/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

2

ASEET Advanced Ada Workshop

Software Engineering Institute

12-15 January 1988

Carnegie Mellon University
Pittsburgh, Pennsylvania

Sponsored by the
U.S. Department of Defense

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ASEET Advanced Ada Workshop, January 12-15, 1988	5. TYPE OF REPORT & PERIOD COVERED Tutorial
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Ada Software Engineering and Education (ASEET) Team	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS Ada Joint Program Office 3E 114, the Pentagon Washington, DC 20301-3081	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office 3E 114, The Pentagon Washington, DC 20301-3081	12. REPORT DATE 12-15 January, 1988
	13. NUMBER OF PAGES 404
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) Ada Joint Program Office	15. SECURITY CLASS (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Joint Program Office, AJPO, Ada Education and Training

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Advanced Ada Workshop is offered semi-annually by the ASEET Team. This document contains the tutorials for the Workshop held January 12-15, 1988 at Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. Topics are Introduction, Software Engineering, Packages, Exceptions, Tasking and Generics.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73 S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Software Engineering Institute

Welcome to the Software Engineering Institute. I'd like to extend my greetings and express the hope that your visit is informative, pleasant, and productive.

If this is your first contact with the SEI, you may be interested in the following background information. The Software Engineering Institute is a federally funded research and development center (FFRDC). Our organization was formed in 1984 in response to the need for advances across all phases of the software engineering process. It is operated by Carnegie Mellon University, under contract with the Department of Defense. Our main directives include bringing the ablest professional minds and the most effective technology to bear on the rapid improvement of the quality of operational software in mission-critical computer systems, exploring and disseminating technology, and establishing standards of excellence for software engineering practice.

We concentrate most of our effort on technology transition, although we are actively involved with technology generation as well. Our approach is to shift software engineering from a labor-intensive basis to a technology-intensive basis through automation based on sound models and theories, and to concentrate on technology transition throughout the managerial, professional, legal, economic, and computational facets of software engineering. Programs at the SEI provide a framework for coordinated efforts within defined areas of technology. They build a foundation to support continued improvement in an area of technology, to develop SEI expertise, and to facilitate the transition of technology and information into practice.

I hope your visit exceeds your expectation.

Sincerely,

Larry Druffel

Larry E. Druffel
Director

Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
(412) 268-7700

Ada Software Engineering Education and Training (ASEET) Team Advanced Ada Workshop Software Engineering Institute 12-15 January 1988

Table of Contents

ADVANCED Ada WORKSHOP
Software Engineering Institute
12-15 January 1988

SCHEDULE

Tuesday, 12 January 1988

		Training Room A
8:00	Welcoming Remarks	
8:15	Introduction	Major Allan Kopp AJPO Representative
9:00	Tutorial - Software Engineering	Capt. Roger Beauman-Keesler AFB
10:00	Break	
10:15	Tutorial - Software Engineering	Capt. Roger Beauman - Keesler AFB Capt. Michael Simpson - Keesler AFB
12:00	Lunch	
1:30	Tutorial - Software Engineering	Capt. Roger Beauman - Keesler AFB Capt. Michael Simpson - Keesler AFB
3:00	Break	
3:15	Tutorial - Software Engineering	Capt. Roger Beauman - Keesler AFB Capt. Michael Simpson - Keesler AFB
5:00	End of Session	
7:00 – 9:00	Birds of a Feather	AJPO activities Major Allan Kopp

WEDNESDAY, 13 JANUARY 1988

		Training Room A
8:30	Tutorial - Packages	Mr. John Bailey - IDA Consultant
10:00	Break	
10:15	Tutorial - Packages	Mr. John Bailey - IDA Consultant
12:00	Lunch	
1:30	Tutorial - Exceptions	Major Pat Lawlis - AFIT
3:00	Break	
3:15	Tutorial - Exceptions	Major Pat Lawlis - AFIT
5:00	End of Session	
7:00 - 9:00	Birds of a Feather	Ada Information Clearinghouse and ASEET Materials Library

Thursday, 14 January 1988

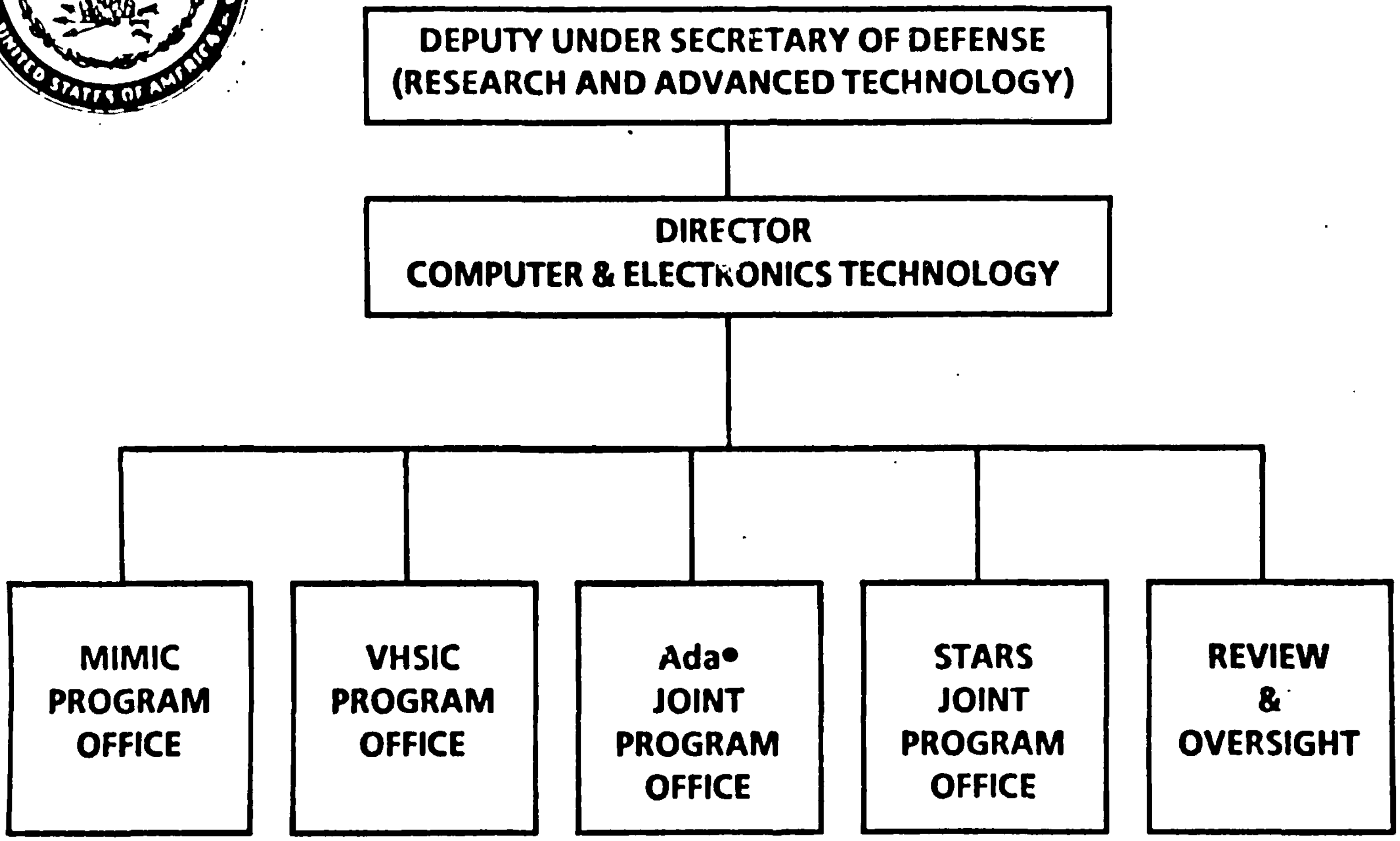
		Training Room
8:30	Tutorial - Tasking	Capt. David Cook - Air Force Academy
10:00	Break	
10:15	Tutorial - Tasking	Capt. David Cook - Air Force Academy
12:00	Lunch	
1:30	Tutorial - Tasking	Capt. David Cook - Air Force Academy
3:00	Break	
3:15	Tutorial - Tasking	Capt. David Cook - Air Force Academy
5:00	End of Session	
7:00 - 9:00	Birds of a Feather	Compilers Ada Tools SIMTEL20

Friday, 15 Janaury 1988

		Training Room A
8:30	Tutorial - Generics	LCDR Lindy Moran - US Naval Academy Major Chuck Engle - SEI
10:00	Break	
10:15	Tutorial - Generics	LCDR Lindy Moran - US Naval Academy Major Chuck Engle - SEI
12:00	Lunch	
1:30	Tutorial - Generics	LCDR Lindy Moran - US Naval Academy Major Chuck Engle - SEI
3:00	Break	
3:15	Tutorial - Generics	LCDR Lindy Moran - US Naval Academy Major Chuck Engle - SEI
5:00	End of Session	

Ada®
PROGRAM

®Ada IS A REGISTERED TRADEMARK
OF THE U.S. GOVERNMENT
(Ada JOINT PROGRAM OFFICE)



•Ada IS A REGISTERED TRADEMARK OF THE U.S. GOVERNMENT (Ada JOINT PROGRAM OFFICE).

OUTLINE

BACKGROUND

PROJECTS

SUMMARY

BACKGROUND

Ada® REQUIREMENTS DEFINED IN A SERIES OF DRAFT SPECIFICATIONS

- STRAWMAN (1975)
  - PRELIMINARY REQUIREMENTS TO EVOKE COMMENT
- WOODENMAN (1975)
  - RESULT OF A FOUR MONTH REVIEW OF STRAWMAN
- TINMAN (1976)
  - FIRST COMPLETE SET OF REQUIREMENTS BASED ON MILITARY INPUTS
- IRONMAN (1977)
  - USED AS BASIS FOR Ada® EFFORT
- STEELMAN (1978)
  - BASED ON RESULTS OF FIRST Ada® PHASE
  - CURRENT STATEMENT OF REQUIREMENTS

STEELMAN REQUIREMENTS

- **STRONG TYPING** — EXPLICIT DEFINITION AND ENFORCEMENT OF CHARACTERISTICS OF DATA ELEMENTS
- **ENCAPSULATION** — RESTRICTS VISIBILITY AND USE OF SELECTED VARIABLES; FACILITATES BOTH TOP DOWN DEVELOPMENT AND ACCUMULATION OF REUSABLE MODULES
- **GENERIC FACILITY** — PROVIDES EXTENSIBILITY TO THE PROGRAMMER WITHOUT EXTENDING THE LANGUAGE
- **TASKING** — STRUCTURED APPROACH TO CONCURRENT PROCESSING AND INTERPROCESS COMMUNICATION
- **EXCEPTION HANDLING** — FACILITY FOR DEALING WITH EXCEPTIONAL SITUATIONS WHICH OCCUR DURING PROGRAM EXECUTION
- **INTERRUPT HANDLING** — FACILITY FOR PROCESSING INTERRUPTS AND OTHER EXTERNAL STIMULI
- **NUMERIC PRECISION** — MACHINE INDEPENDENT APPROACH TO INTEGERS, FIXED POINT AND FLOATING POINT
- **MACHINE DEPENDENCIES** — EXPLICIT DECLARATION AND ENCAPSULATION OF HARDWARE AND OPERATING SYSTEM DEPENDENCIES

LANGUAGES FORMALLY EVALUATED

- COBOL
- FORTRAN
- PL/I
- TACPOL
- HAL/S
- CMS-2
- CS-4
- JOVIAL J3B
- ALGOL 60
- CORAL 66
- SIMULA 67
- ALGOL 68
- PASCAL
- LIS
- LTR
- JOVIAL J73
- SPL/I
- EUCLID
- MORAL
- ECL
- PDL/2
- PEARL
- RTL/2

Ada®
STANDARDIZATION

REFERENCE MANUAL PUBLISHED	JULY 1980
MIL-STD 1815 DESIGNATED	DECEMBER 1980
ANSI CANVASS INITIATED	APRIL 1981
ANSI CANVASS COMPLETED	OCTOBER 1981
ANSI RECANVASS INITIATED	JULY 1982
ANSI RECANVASS COMPLETED	SEPTEMBER 1982
ANSI/MIL-STD 1815A Ada®	JANUARY 1983

HIGH ORDER LANGUAGE COMPARISON

Evolution ——————————>

Requirements:	FORTRAN	JOVIAL	PL/1	PASCAL	ADA
Arithmetic Expressions	X	X	X	X	X
Program Structure	X	X	X	X	X
Iteration Loops	X	X	X	X	X
Formatted Input/Output	X	-	X	-	X
Block Structure		X	X	X	X
Recursion		X	X	X	X
Formal Syntax Definition		X	X	X	X
Multi-Tasking			X	-	X
Exception Handling			X	-	X
Extensive Typing Mechanisms			X	X	X
Pointers			X	X	X
Strong Typing				X	X
Enumeration Types				X	X
Subrange Types				X	X
Type Declarations				X	X
Aggregate					X
Derived Type					X
Exception					X
Pragma					X
Generic Program Unit					X
Rendevous					X

ADA CAPABILITIES VS OTHER DOD HOLS

	FOR	CMS	J73	ADA	MAX
DESIGN CRITERIA	38	33	43	51	59
GENERAL SYNTAX	28	28	33	35	38
DATA TYPING	46	65	88	104	105
CONTROL	31	34	48	50	51
FUNCTIONS & I/O	11	22	30	37	37
REAL TIME PROCESSING	0	1	4	45	48
OTHER TECHNIQUES	17	30	47	59	67
TOTALS	171	213	293	381	405

PROGRAM ELEMENT 63226F

TITLE: DOD COMMON PROGRAMMING LANGUAGE (Ada)

DOD MISSION AREA: ELECTRONIC AND PHYSICAL SCIENCES

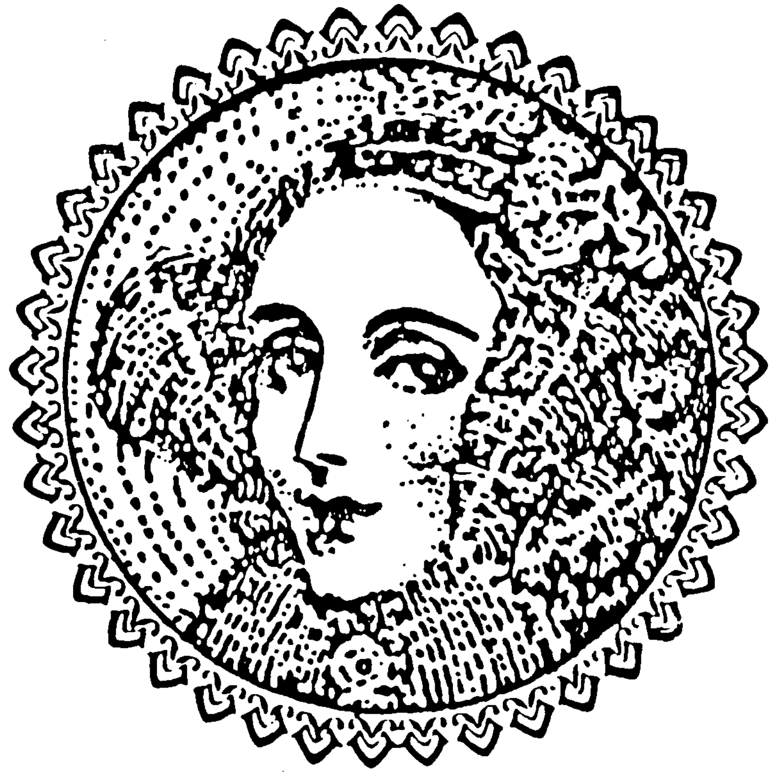
BUDGET ACTIVITY: ADVANCED TECHNOLOGY DEVELOPMENT

DESCRIPTION:

- PART OF DOD ACTIVITY TO INTRODUCE, IMPLEMENT AND PROVIDE LIFE CYCLE SUPPORT FOR Ada
- PROVIDE RESOURCES TO MEET THOSE LANGUAGE SUPPORT REQUIREMENTS WHICH ARE COMMON TO THE VARIOUS SERVICES AND AGENCIES
- PROVIDE FOR CONFIGURATION CONTROL OF THE Ada LANGUAGE
- PROVIDE FOR STANDARDIZATION

OPR: Ada JOINT PROGRAM OFFICE (AJPO)

PROJECTS



AJPO PROJECTS

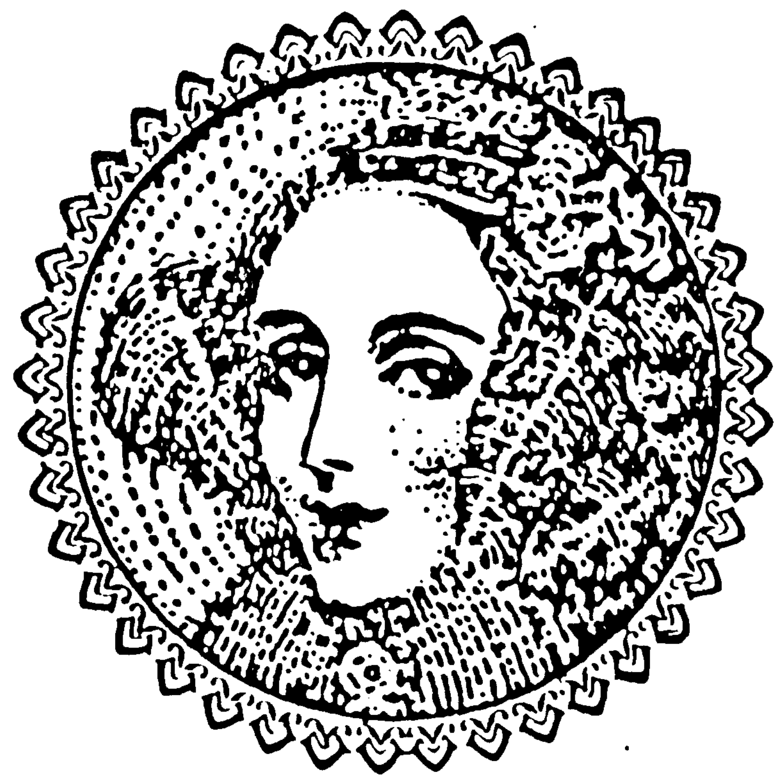
LANGUAGE CONTROL

VALIDATION

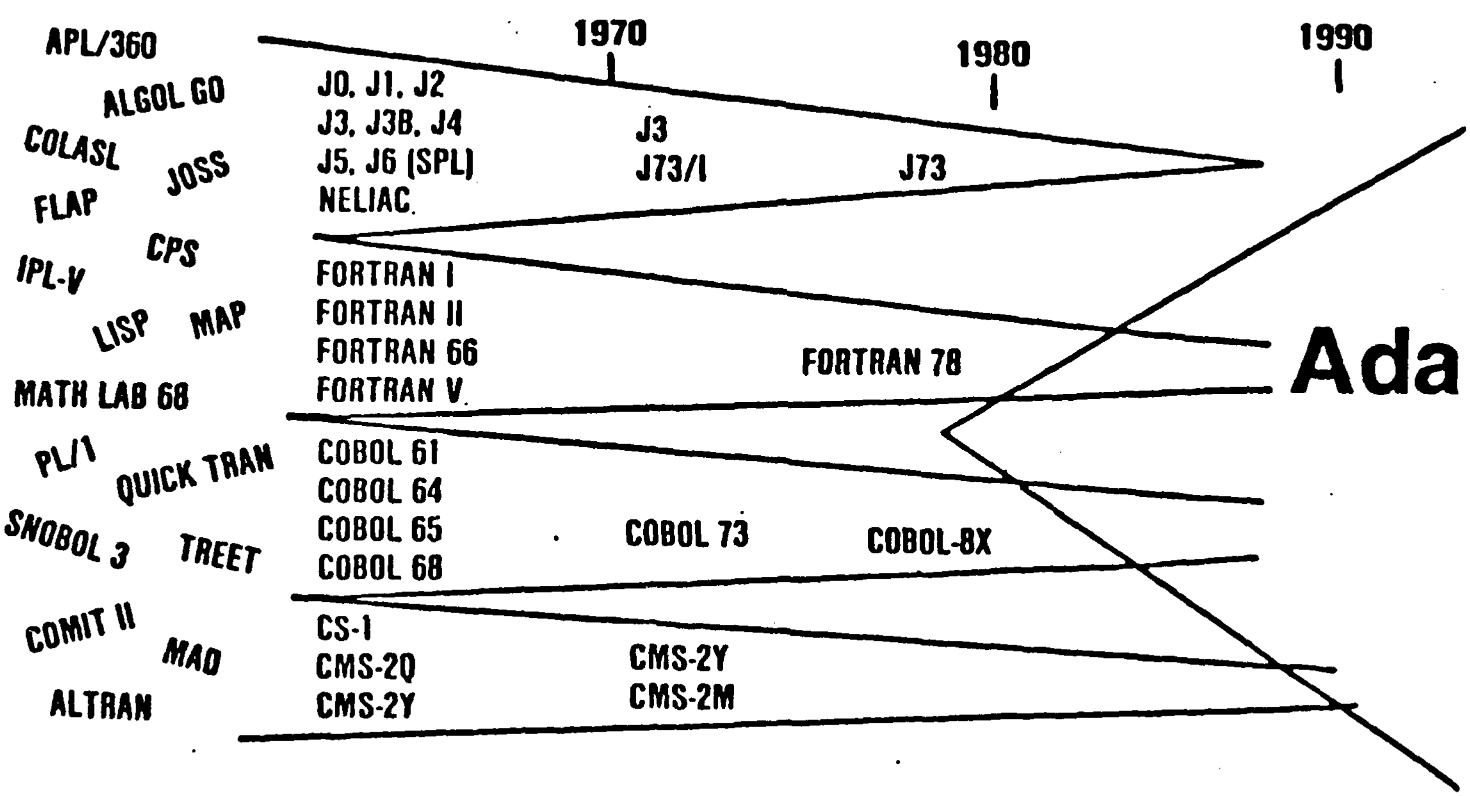
EDUCATION & PROMOTION

Ada PROGRAMMING SUPPORT ENVIRONMENTS

TECHNOLOGY INSERTION



LANGUAGE CONTROL



LANGUAGE CONTROL

- Ada BOARD
- AMERICAN NATIONAL STANDARDS INSTITUTE (ANSI)
- FEDERAL INFORMATION PROCESSING STANDARD (FIPS)
- INTERNATIONAL STANDARDS ORGANIZATION (ISO)
- NYU Ada ED
- TRADEMARK REGISTRATION/CERTIFICATION
- Ada FORMAL METHODS
- Ada RATIONALE
- REVISED LANGUAGE REFERENCE MANUAL

VALIDATED Ada

THIS PRODUCT CONFORMS TO ANSI/MIL-STD-1815A AS DETERMINED BY THE AJPO UNDER ITS CURRENT TESTING PROCEDURES

LANGUAGE CONTROL

AJPO PROGRAM SUPPORT:

- Establishing & Maintaining Ada as MILITARY, ANSI & FIPS standard permits explicit designation in RFPs & SOWs

- Trademark establishment & registration actions protect program offices from marketing of invalid vendor products

- LRM, Ada/ED and Rationale clarify standard for compiler implementers allowing rapid development & tailoring to meet program specific needs

MILITARY USE

EMBEDDED COMPUTER SYSTEMS 1975 -

ORGINAL TARGET USE FOR Ada INTENDED BY HOLWG

INCLUDES ALL DEFENSE SYSTEMS WHERE COMPUTERS ARE A PART OF A LARGER SYSTEM; MISSILES, A/C, SHIPS, TANKS.

EXPECTED APPLICATIONS INCLUDE:

TRACKING, SYSTEMS SW, NAVIGATION, COMMAND, CONTROL & COMMUNICATIONS, SIMULATION, COMPUTATION

EXAMPLES: ATF, CAMP

MILITARY USE

MISSION CRITICAL COMPUTER RESOURCES 1983 -

DIRECTED BY USDRE MEMO; EFFECTIVE 1 Jul 84

INCLUDES EMBEDDED SYSTEMS AND ALL ADP SYSTEMS WHERE THE FUNCTION, OPERATION, OR USE INVOLVES

INTELLIGENCE, CRYPTOLOGIC, COMMAND, CONTROL OR CRITICAL TO MISSION FULFILLMENT

EXAMPLES: MILSTAR, AFATDS

EMBEDDED COMPUTER SYSTEMS 1975 -

MILITARY USE

INFORMATION SYSTEMS 1984 -

DIRECTED BY ARMY INFORMATION SYSTEMS COMMAND, EFFECTIVE 1 OCT 1984

FIPS APPROVED OCT 1985 OPENING Ada TO AIR FORCE & NAVY INFORMATION SYSTEMS

INCLUDES ROUTINE ADMIN AND BUSINESS APPLICATIONS; E.G.,

PAYROLL, FINANCE, LOGISTICS AND PERSONNEL MANAGEMENT

EXAMPLES: ISS, AF PHASE IV

MISSION CRITICAL COMPUTER RESOURCES 1983 -

EMBEDDED COMPUTER SYSTEMS 1975 -

VALIDATION

VALIDATION APPROACH

POLICY & PROCEDURES

Ada COMPILER
VALIDATION CAPABILITY

Ada VALIDATION ORGANIZATION

Ada VALIDATION FACILITY

VALIDATION

Ada VALIDATION PROCEDURES AND GUIDELINES

Ada VALIDATION ORGANIZATION

Ada VALIDATION FACILITIES - WRIGHT-PATTERSON AFB
- GSA/NBS
- GERMANY
- FRANCE
- UNITED KINGDOM

Ada COMPILER VALIDATION CAPABILITY

AUTOMATED VALIDATION TOOLS

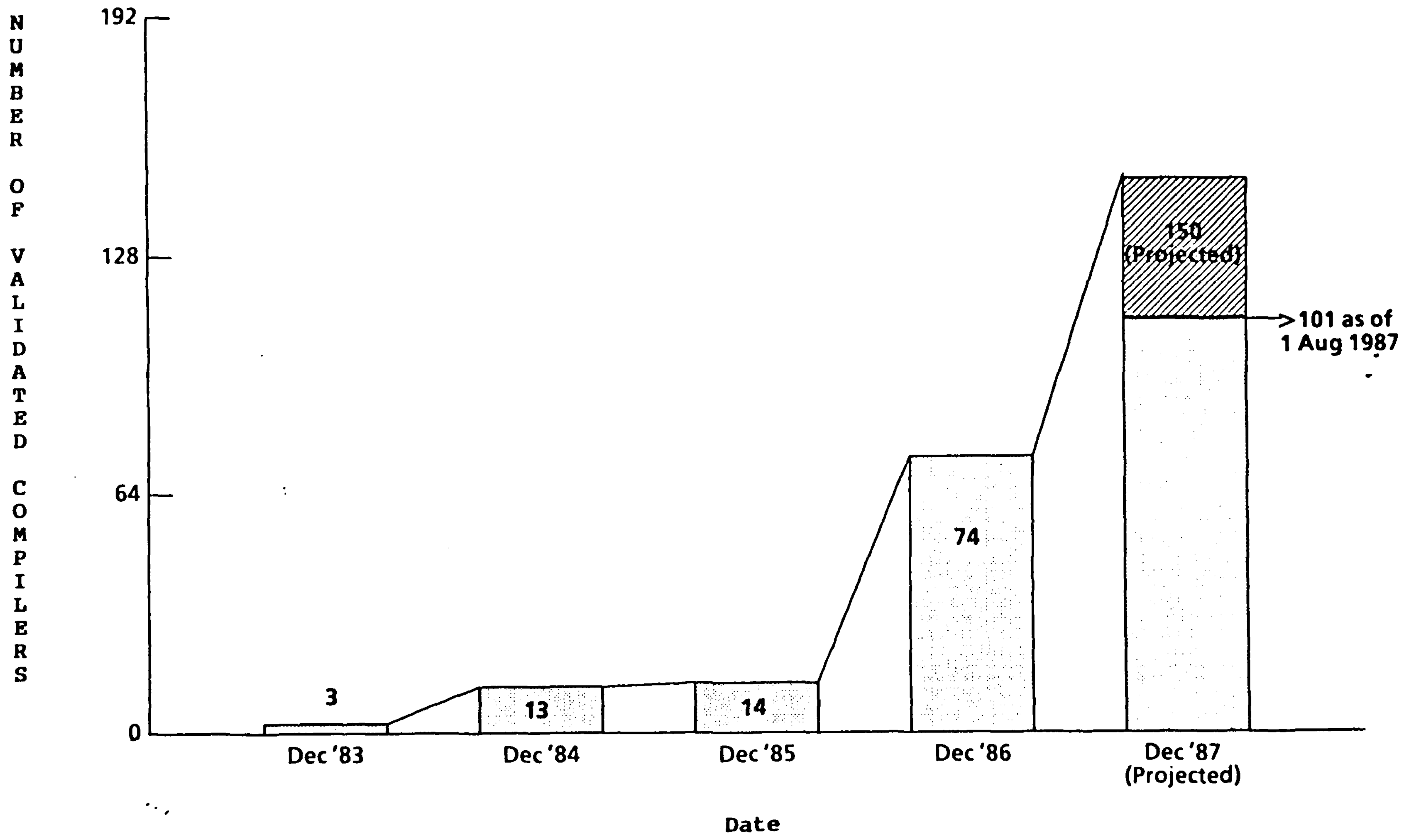
VALIDATION

AJPO PROGRAM SUPPORT:

- Certification System with DoD Ada Validation Facility (at WPAFB) permits program office enforcement of Ada standard

- Ada Compiler Validation Procedures and Guidelines coordinates contractor and program office software acquisition/maintenance actions with Ada validation

Validated Ada Compilers (1983 - 1987)



EDUCATION AND PROMOTION

EDUCATION

- Ada SOFTWARE ENGINEERING EDUCATION AND TRAINING (ASEET) TEAM
  - ASEET ANNUAL SYMPOSIUM
  - ADVANCED Ada WORK SHOPS
  - PROFESSIONAL DEVELOPMENT BRIEFINGS
  - ASEET MATERIALS LIBRARY
  - TRAINING GUIDE FOR Ada SOFTWARE ENGINEERING PROJECTS
  - ASEET PUBLIC REPORTS
- AFCEA STUDY
- CATALOG of RESOURCES in EDUCATION for Ada SOFTWARE ENGINEERING

PROMOTION

Ada INFORMATION CLEARINGHOUSE PRODUCTS

- ON-LINE Ada-INFORMATION DIRECTORY
- CLEARINGHOUSE STAFF AVAILABLE FOR TELEPHONE QUERIES
- CATALOG OF RESOURCS FOR EDUCATION IN Ada AND SOFTWARE ENGINEERING (CREASE 4.0)
- CLEARINGHOUSE STAFF AVAILABLE FOR TELEPHONE QUERIES
- Ada BIBLIOGRAPHY VOL III
- DOCUMENTS REFERENCE LIST
- VALIDATION COMPILER LIST
- Ada IMPLEMENTATIONS LIST
- CALENDAR OF Ada EVENTS
- DOCUMENT SEARCHES
- SPECIAL TOPIC PACKETS
  - GENERAL INFO
  - EDUCATION
  - VALIDATION
  - HISTORICAL
  - CURRENT AWARENESS

PROMOTION (CON'T)

RECENT ACTIVITIES

Ada INFORMATION CLEARINGHOUSE	ONGOING, NEW CONTRACT EARLY 1987
Ada USAGE DATA BASE	INITIATED FY86, CONTINUALLY EXPANDING
PRODUCTS & TOOLS DATABASE	INITIATED FY86, CONTINUALLY EXPANDING
DDN SUPPORT & PUBLIC BULLETIN BOARD	ONGOING COMMUNCIATIONS SERVICE

PROMOTION (CON'T)

AdaIC MONTHLY ACTIVITY*

INCOMING CALLS	310
SPECIAL TOPIC PACKETS	155
INDIVIDUAL DOCUMENTS	140
ELECTRONIC BULLETION BOARD	
DOCUMENTS DOWNLOADED	3,643
CALLS PER MONTH	529
NEWSLETTER (QUARTERLY)	4,000

*(average for three month period 4/87-6/87)

EDUCATION & PROMOTION

AJPO PROGRAM SUPPORT:

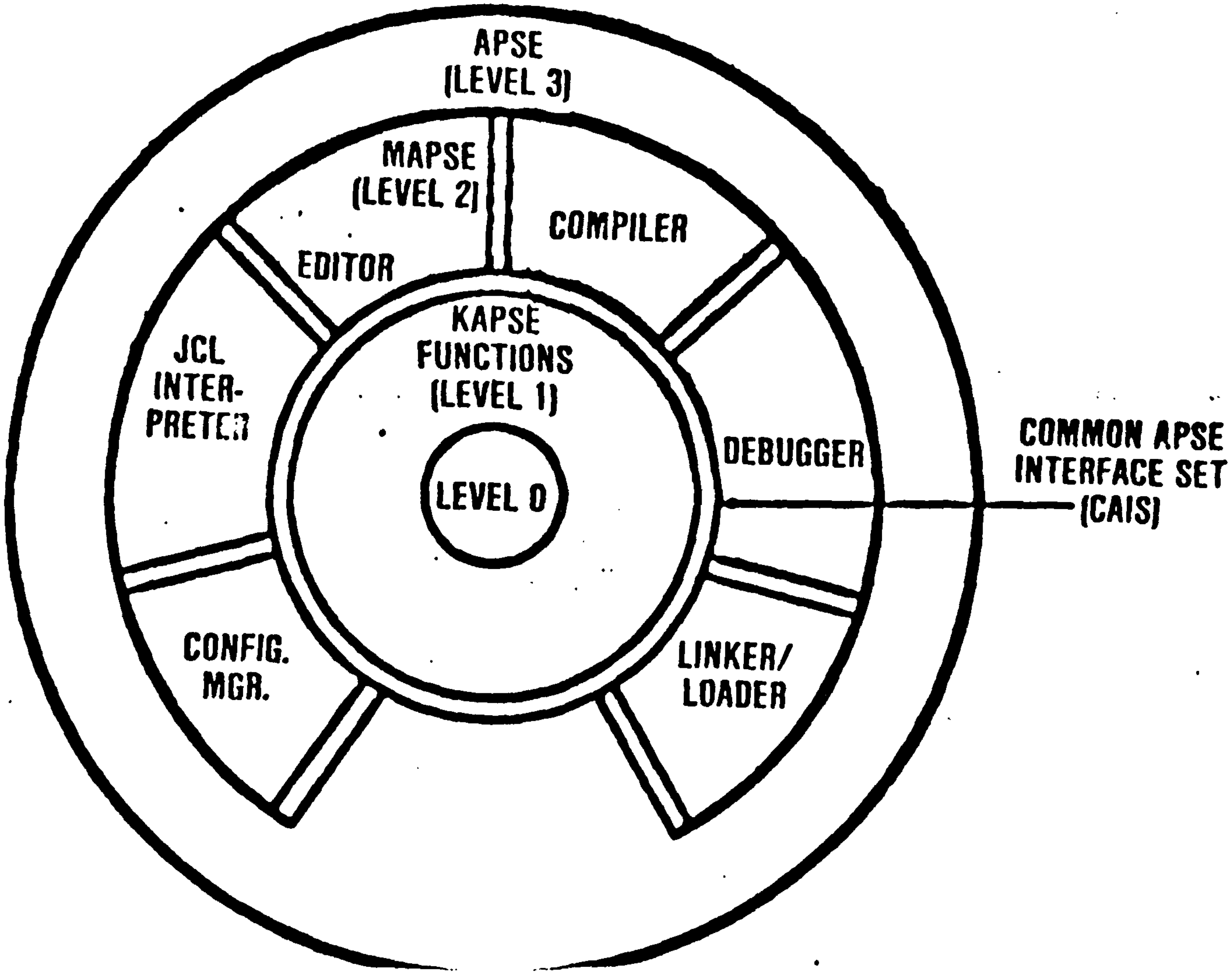
- ASEET coordinates DoD training and education activities which support program office personnel and higher level management

- AdaIC provides up to date information to program offices on the availability of Ada technologies for use on DoD systems

COOPERATIVE ACTIVITIES RELATED TO Ada

CONGRESSIONAL INITIATIVES:	NUNN AMENDMENT, SOFTWARE VALLEY
DoD PROGRAMS:	Ada, STARS, SEI, VHSIC, SDI
MILITARY STANDARDS:	MIL-STD-2167 & HANDBOOK, MIL-STD-2168
GOVERNMENT THRUSTS:	NASA, NBS, GSA
INDUSTRY GROUPS:	SIGAda, AdaJUG, AFCEA, IEEE
ACADEMIC ACTIVITIES:	Ada TECHNOLOGY CENTER, EDUCATIONAL SYMPOSIUM

Ada PROGRAMMING SUPPORT ENVIRONMENTS



Ada PROGRAMMING SUPPORT ENVIRONMENTS

- **APSE PERFORMANCE**

  EVALUATION & VALIDATION (E&V) TEAM
  Ada COMPILER BENCHMARKS
  Ada COMPILER EVALUATION CAPABILITY
  Ada RUN TIME ENVIRONMENT WORKING GROUP (ARTEWG)

- **PORTABILITY OF TOOLS, APPLICATIONS & DATA BASES**

  KAPSE INTERFACE TEAM (KIT)
  COMMON APSE INTERFACE SET (DoD-STD-CAIS)
  CAIS VALIDATION CAPABILITY
  CAIS OPERATIONAL DEFINITION

SOFTWARE DEVELOPMENT PRODUCTIVITY

IMPROVEMENTS WITH Ada

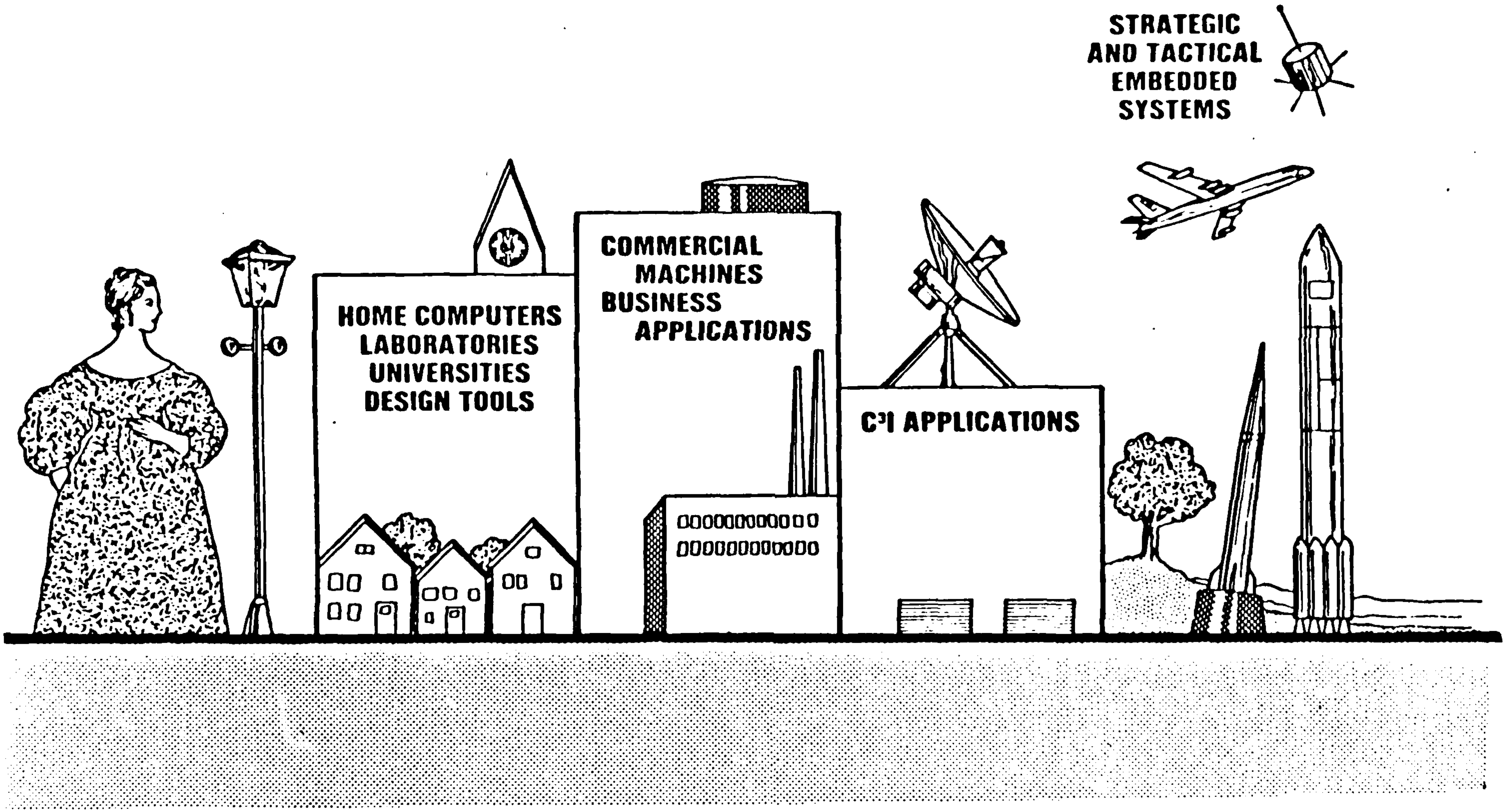
- LANGUAGE FEATURES SUPPORTING REUSE
- PUBLICLY AVAILABLE SOFTWARE INVENTORIES
- LIFE CYCLE TOOLS
- NETWORKED PROGRAMMER WORKSTATIONS
- RAPID PROTOTYPING
- EMPHASIS ON SOFTWARE ENGINEERING
- REUSEABLE SOFTWARE COMPONENTS
- INTEGRATED ENVIRONMENTS
- OOD & OTHER METHODOLOGIES

Ada PROGRAMMING SUPPORT ENVIRONMENTS

AJPO PROGRAM SUPPORT:

- AJPO supported initial Air Force and Army APSE Developments for multiple reuse in systems acquisitions
- Prototype Ada Compiler Benchmarking system and Ada Compiler Evaluation Capability provide program offices with technologies to evaluate Ada compiler performance
- Assisting program offices in requesting vendors to correct/improve Ada Technology performance (SAC)
- Sponsoring ARTEWG to focus industry on realtime performance issues to support Ada use in embedded weapons

TECHNOLOGY INSERTION



HAVE Ada® OPPORTUNITIES BEEN KNOCKING?

Ada® IS A TRADEMARK OF THE U.S. GOVERNMENT (Ada JOINT PROGRAM OFFICE)

TECHNOLOGY INSERTION

- DoD DIRECTIVES 3405.1 AND 3405.2
- Ada TECHNOLOGY INSERTION PROGRAM (ATIP)
- NATO INITIATIVE (NUNN AMENDMENT)

DOD DIRECTIVE 3405.1 -- APRIL 2, 1987

COMPUTER PROGRAMMING LANGUAGE POLICY

'UMBRELLA' POLICY FOR ALL DOD -- (REPLACES DODI 5000.31)

ESTABLISHES LONG-RANGE GOAL OF:
- TRANSITION TO THE USE OF ADA FOR ALL DOD SOFTWARE DEVELOPMENT

MANDATES ADA FOR:
- INTELLIGENCE SYSTEMS -- (INCLUDES MCCR)
- COMMAND & CONTROL OF MILITARY FORCES -- (INCLUDES MCCR)
- SYSTEMS INTEGRAL TO A WEAPON SYSTEM -- (DODD 3405.2)

DODD 3405.1 CONTINUED

MANDATES ADA FOR ALL OTHER (MIS) APPLICATIONS EXCEPT:

- WHERE ANOTHER APPROVED HOL IS MORE COST EFFECTIVE OVER THE APPLICATION'S LIFE-CYCLE

UPDATES 'APPROVED' HOL LIST:

ADA	C/ATLAS	COBOL	
CMS-2M	CMS-2Y	FORTRAN	
JOVIAL (J73)	MINIMAL BASIC	PASCAL	SPL/1

DOD DIRECTIVE 3405.2
MARCH 30, 1987

USE OF ADA IN WEAPON SYSTEMS

MANDATES ADA FOR ALL SYSTEMS INTEGRAL TO WEAPON SYSTEMS, MEANING:

- PHYSICALLY A PART OF, DEDICATED TO, ESSENTIAL IN REAL TIME
- USED FOR SPECIALIZED TRAINING, DIAGNOSTIC TESTING & MAINTENANCE
- USED FOR SIMULATION, CALIBRATION OR RESEARCH & DEVELOPMENT

APPLIES TO ALL PHASES OF THE LIFE CYCLE AND MAJOR UPGRADES

DODD 3405.2 CONTINUED

REQUIRES USE OF:

- VALIDATED COMPILERS
- SOFTWARE ENGINEERING PRINCIPLES (2167/HANDBOOK)
- ADA-BASED PROGRAM DESIGN LANGUAGE

REQUIRES DOD COMPONENTS TO DESIGNATE AN

- ADA EXECUTIVE OFFICAL AND AN
- ADA WAIVER CONTROL OFFICER

REQUIRES COMPONENT ADA IMPLEMENTATION PLAN BY 30 AUG 87

TECHNOLOGY INSERTION (CON'T)
ATIP PROGRAM

- BREAKDOWN TECH RISK BARRIERS
  - MULTILEVEL SECURITY
  - DISTRIBUTED PROCESSING
  - COMPLEX QUERIED DBMS
  - ADVANCED ARCHITECTURES
  - ULTRA HIGH PERFORMANCE CODE
  - MAXIMUM REUSABLE COMPONENTS
- DIRECT PROGRAM OFFICE ASSISTANCE
  - SIMULATION
  - AVIONICS
  - FIRE CONTROL
  - MISSILES
  - C3I
  - ELECTRONIC WARFARE
- MERIT SELECTION
- COST SHARED
- STANDARD METRICS COLLECTION, ANALYSIS AND REPORTING

NATO SWG ON APSE

PRINCIPLE POCs: Virginia L. Castor, Chair SWG on APSE
David R. Taylor, U.S. Delegate/U.S. PM

BRIEF TASK DESCRIPTION:

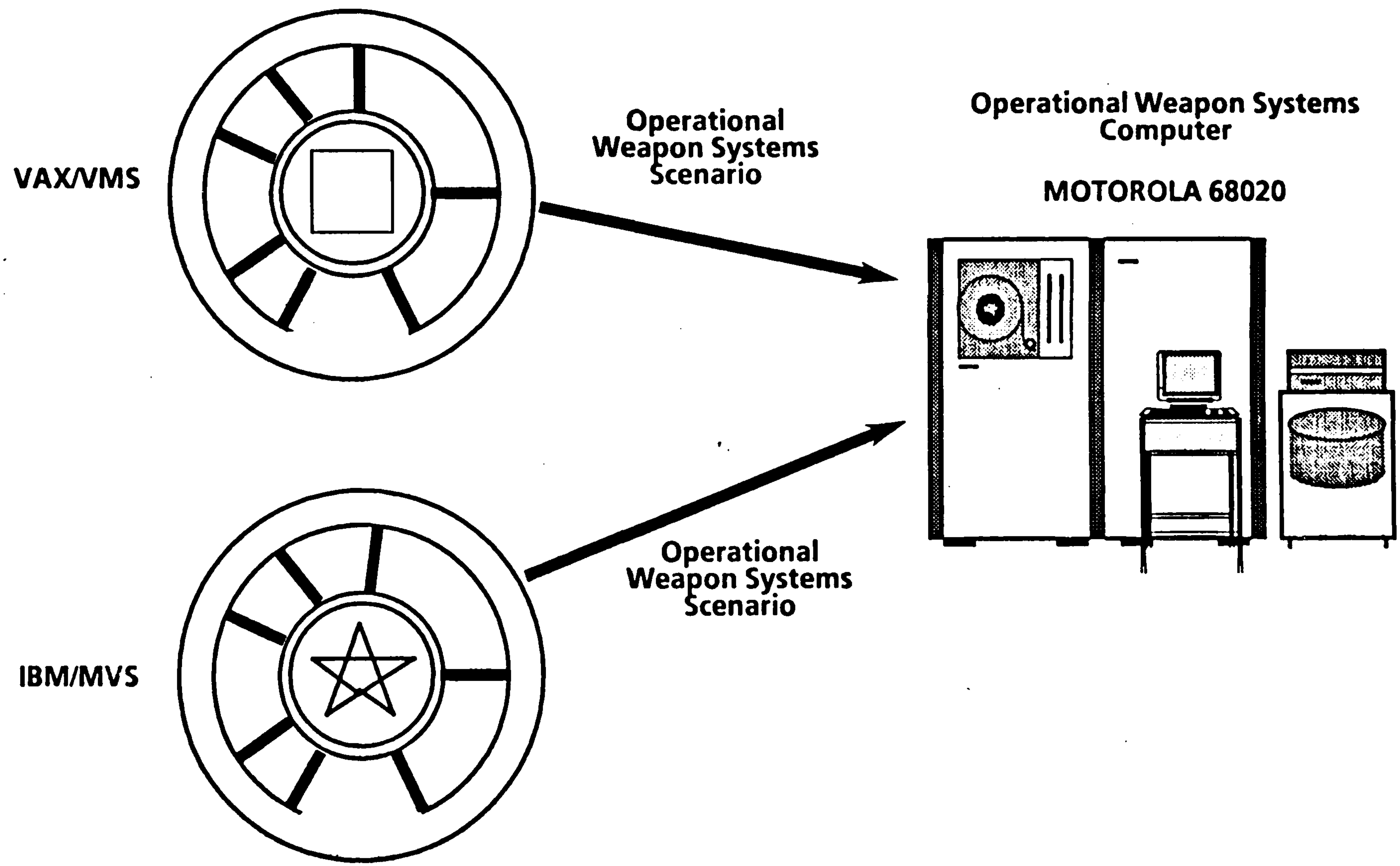
- The Special Working Group on Ada Programming Support Environments (SWG on APSE), consisting of representatives from Belgium, Canada, Denmark, France, Germany, Italy, Netherlands, Norway, Spain, United Kingdom, and the United States, have agreed to a memorandum of understanding (MOU) to:

a. develop and demonstrate a group of software tools representative of a usable APSE through their initial implementation on two distinct computer architectures using an agreed interface set;

b. develop methods and tools for the evaluation of APSEs and demonstrate this technology on the products resulting from this effort; and

c. develop the requirements and specifications of an interface standard for APSEs, based on reviews of evolutionary interface developments to be recommended for adoption and use by NATO and the participating nations.

DEMONSTRATION OF APSE CAPABILITY



Ada TECHNOLOGY INSERTION

AJPO PROGRAM SUPPORT:

- Expanding Acquisition/Maintenance Management structure to support Ada in DoD Directives and Mil-Std-2167

- Providing Direct Assistance to Program offices:

  - SDIO (FIXIT)
  - ATF (consultation)
  - 155 MM HIP (AdaTAG)
  - Flight Dynamics Lab (F-15 DEMO)
  - Phase IV (funded Ada evaluation ECP)
  - Microprocessor Evaluation - (Air Force MIS Application)
  - Assisting FAA in selecting Ada for Advanced Automated System
  - Assisting Dept of Commerce in selecting Ada for use in fexible mfg

SUMMARY

SUMMARY
Ada PROGRAM IMPACTS WITHIN THE DoD

RISING MILITARY USE

STRONGER DISCIPLINE IN LANGUAGE STANDARDIZATION

WIDER COOPERATION

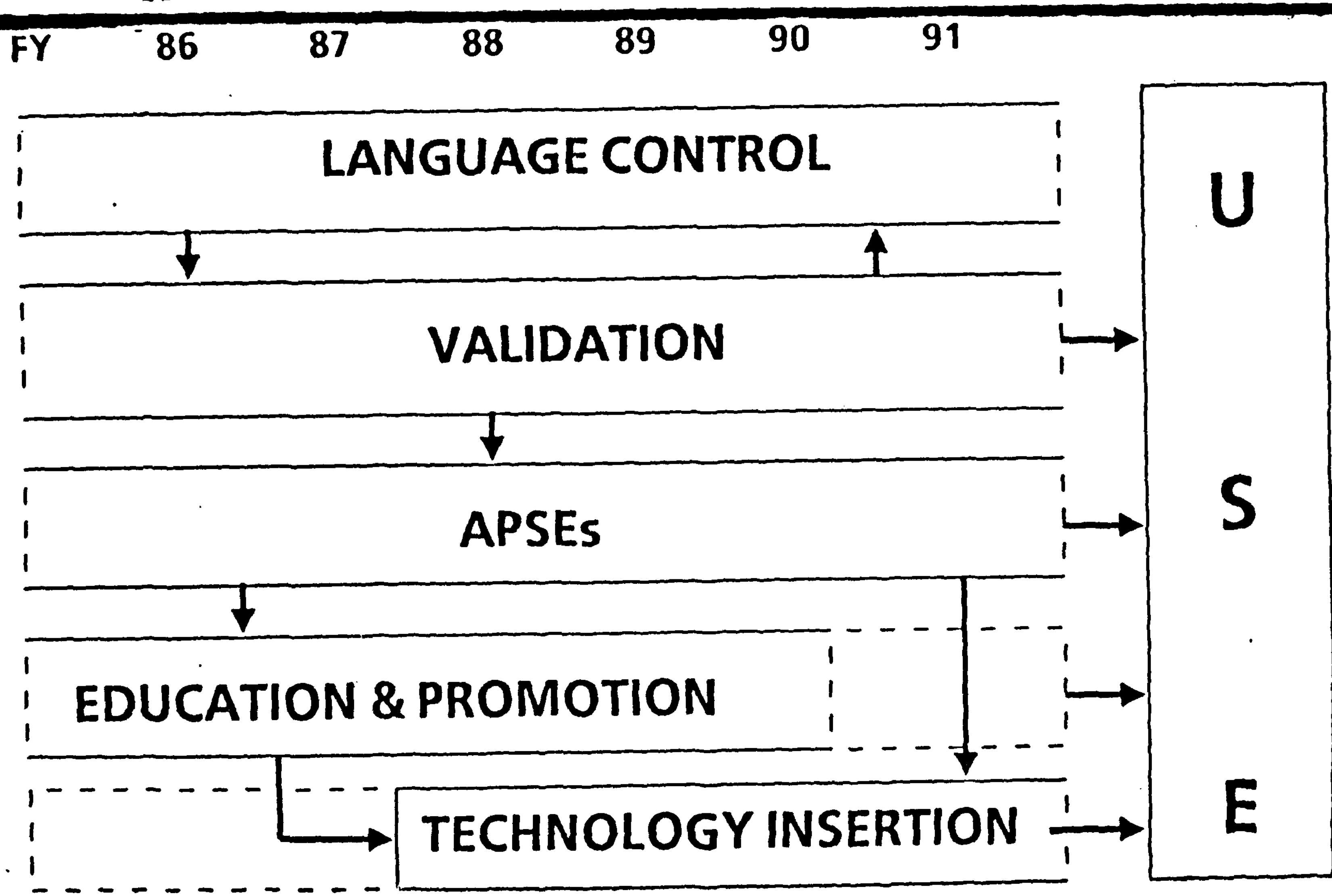
INCREASED AVAILABILITY OF COMPILERS & ENVIRONMENTS

IMPROVED PORTABILITY

HIGHER PRODUCTIVITY

DEMAND FOR PERFORMANCE EVALUATION

Ada PROGRAM ROADMAP



ADVANCED Ada WORKSHOP

Applied Ada Software Engineering

Capt Roger D. Beauman

Capt Michael S. Simpson

Ada Software Engineering Education and Training (ASEET) Team

APPLIED Ada SOFTWARE ENGINEERING

* Basic Problem
  - -- Projection to the 1990's
  - -- A Macro Solution
* A Practical Solution
  - -- Software Engineering
  - -- Ada
* Software Engineering
  - -- Goals
  - -- Principles
* Why Ada ?
  - -- Features of Ada
  - -- Software Engineering Applications

BASIC PROBLEM

Projection to the 1990's

* Multiprocessors - Networks and Parallel Architectures
* Distributed Databases
* Hardware Capabilities
* Software Demands
* Hardware Costs

DISTRIBUTED DATABASES

* Central Control Over Data
* Minimize Effort in Storing Data
* "The Ada Package Store"

DISTRIBUTED DATABASES

* Central Control Over Data
* Minimize Effort in Storing Data
* "The Ada Package Store"

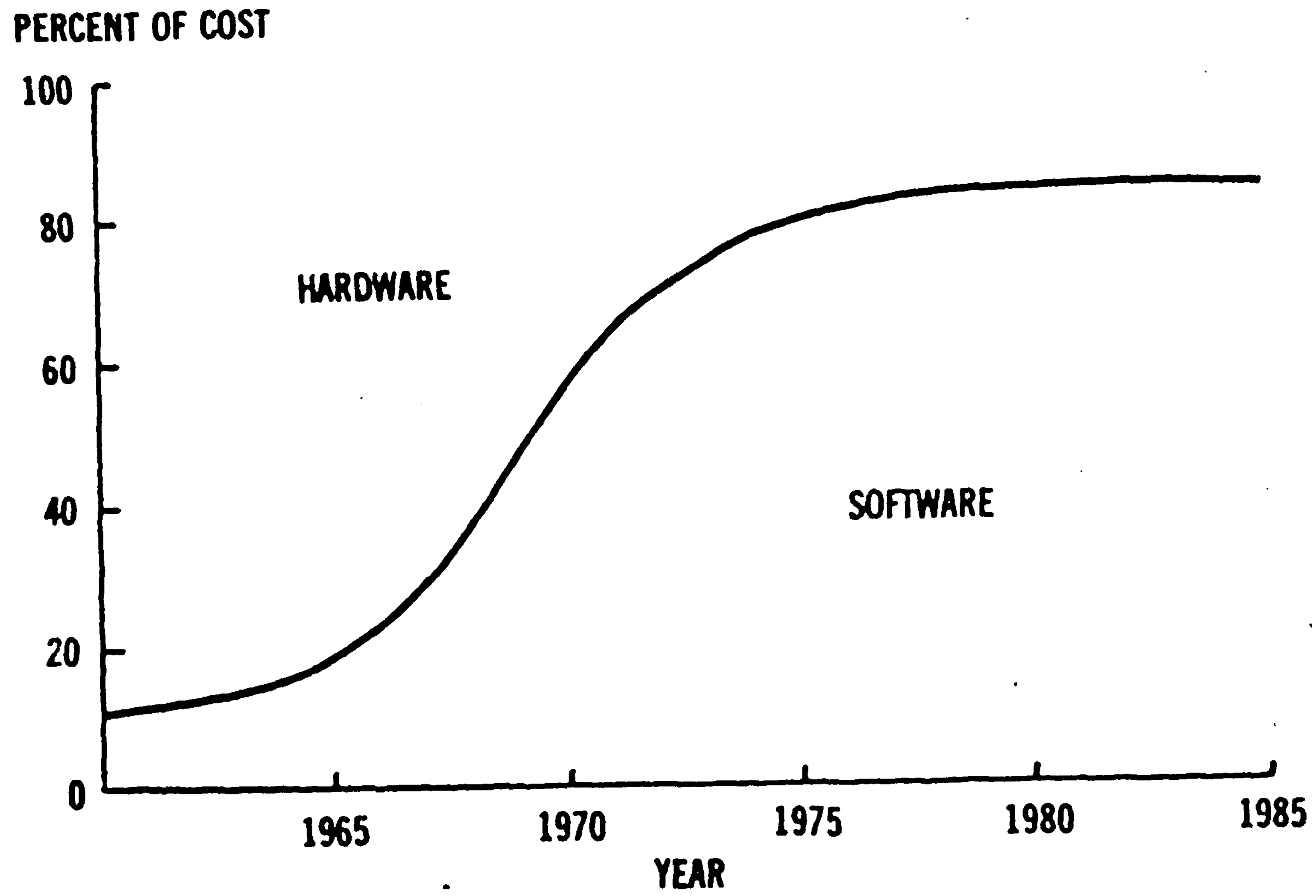
HARDWARE CAPABILITIES

* Mainframe in a Micro
  - -- Intel 80286, 80386, 80486, ???
  - -- Motorola 68000, 68010, 68020, 68030, ???
* Screen Resolution
  - -- Desktop Publishing, CAD/CAM
* Storage Devices
  - -- 100+ MB Hard Disks
  - -- Access Times - 18 ms to 40 ms
* Opens New Fields of Applications

SOFTWARE DEMANDS

* New Users with Consumer Relationships
* Non-Technical Arenas
  - -- Need Guarantees
  - -- Demand Reliability
* Development is the Key
  - -- Design is Paramount
    - --- Simplistic Operations; i.e. TV
  - -- Costs of Errors
  - -- Other Considerations

HARDWARE/SOFTWARE COST TRENDS



A MACRO SOLUTION

* Greater Use of Automation
* Higher Levels of Abstraction
* Reuseability
  -- Isolate Commonality
  -- Create Workable Abstractions
  -- Reuseable Parts Library
* Rapid Prototyping
  -- Gain Insight
  -- Evaluate Design Expectations
  -- Compare Design Alternatives

A solution offered by Edward Lieblein

A PRACTICAL SOLUTION

Software Engineering Myths

* Anyone Can Be a Software Engineer
* Automated Tools = Software Engineering
* Structured Programming = Software Engineering
* Structured Analysis = Software Engineering
* Code Re-use = Software Engineering
* It Will Make Programming Obsolete
* AI Will Make It Effortless
* Fantastic Productivity Gains
* Ada = Software Engineering

SOFTWARE ENGINEERING

A PRACTICAL SOLUTION

* What Is It ?
* Why Is It Needed ?
* The State of the Art
* The State of the Practice
* Why Now ?

CHARACTERISTICS OF DoD SOFTWARE

* Expensive
* Incorrect
* Unreliable
* Difficult to predict
* Unmaintainable
* Not reusable

FACTORS AFFECTING DoD SOFTWARE

* Ignorance of life cycle implications

* Lack of standards

* Lack of methodologies

* Inadequate support tools

* Management

* Software professionals

CHARACTERISTICS OF DoD SOFTWARE REQUIREMENTS

* Large
* Complex
* Long lived
* High reliability
* Time constraints
* Size constraints

THE FUNDAMENTAL PROBLEM

* Our inability to manage the COMPLEXITY of our software systems

* Lack of a disciplined, engineering approach

SOFTWARE ENGINEERING

THE ESTABLISHMENT AND APPLICATION OF SOUND ENGINEERING =>

* Environments
* Tools
* Methodologies
* Models
* Principles
* Concepts

SOFTWARE ENGINEERING

COMBINED WITH =>

* Standards
* Guidelines
* Practices

SOFTWARE ENGINEERING

TO SUPPORT COMPUTING WHICH IS =>

* Understandable
* Efficient
* Reliable and safe
* Modifiable
* Correct

THROUGHOUT THE LIFE CYCLE OF A SYSTEM

SOFTWARE ENGINEERING

* Purposes
* Concepts
* Mechanisms
* Notation
* Usage

PURPOSES

* Create software systems according to good engineering practice

* Manage elements within the software life cycle

CONCEPTS

* Derive the architecture of software systems

* Specify modules of the system

MECHANISMS

* Tools for:
  - Writing operating systems
  - Tuning software
  - Prototyping

* Techniques for:
  - Managing projects
  - Systems analysis
  - Systems design

* Standards for:
  - Coding
  - Metrics
  - Human and machine interfacing

NOTATION

* Languages for writing linguistic models

* Documentation

USAGE

* Embedded systems
* Data processing
* Control
* Expert systems
* Research and development
* Decision support
* Information management

CONTENT AREAS

* Communication skills
* Software development and evolution processes
* Problem analysis and specification
* System design
* Data Engineering
* Software generation
* System quality
* Project management
* Software engineering projects

PROGRAMMING LANGUAGES AND SOFTWARE ENGINEERING

* A programming language is a software engineering tool

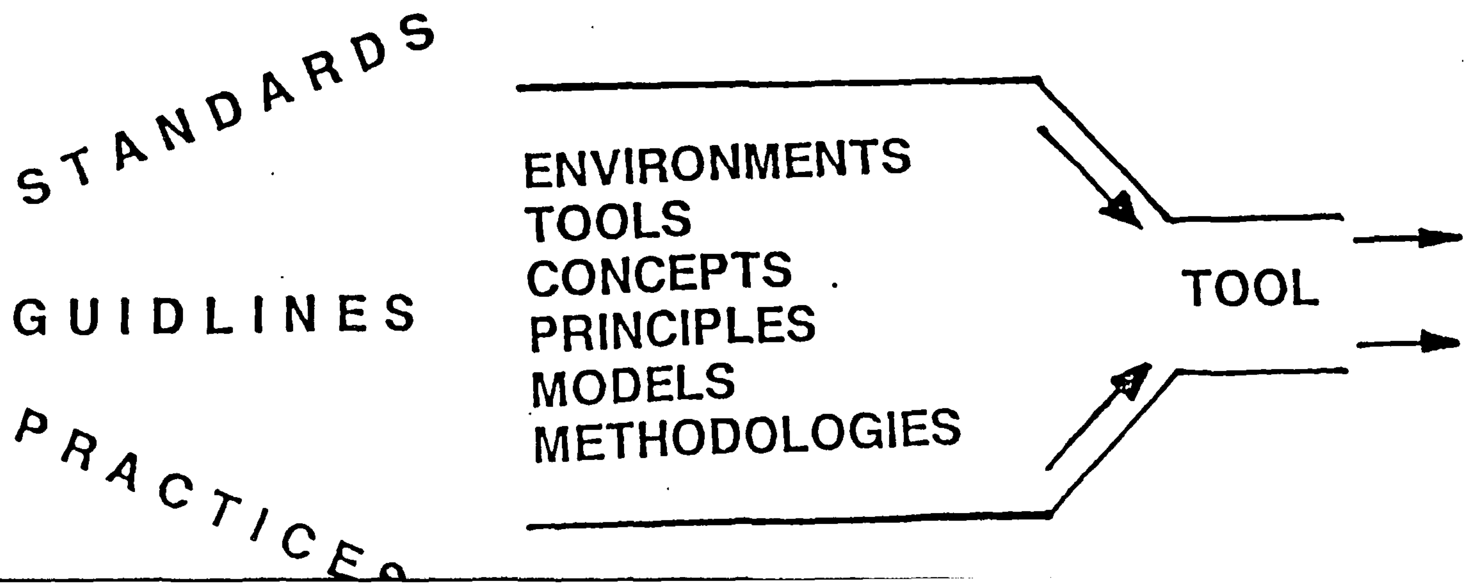
* A programming language EXPRESSES and EXECUTES design methodologies

* The quality of a programming language for software engineering is determined by how well it supports a design methodology and its underlying models, principles, and concepts

TRADITIONAL PROGRAMMING LANGUAGES AND SOFTWARE ENGINEERING

Programming Languages

- Were not engineered
- Have lacked the ability to express good software engineering
- Have acted to constrain software engineering



A PRACTICAL SOLUTION

Ada

Ada and Software Engineering

* They Aren't the Same Thing
* Ada Has Unique Features That Facilitates Software Engineering
* You CAN Write Bad Code in Ada
* Ada is NOT the Total Answer

USAISEC

Ada
AND
SOFTWARE ENGINEERING

Ada

- Was itself "engineered" to support software engineering
- Embodies the same concepts, principles, and models to support methodologies
- Is the best tool (programming language) for software engineering currently available

---

	ENVIRONMENTS	
STANDARDS	TOOLS	T
	CONCEPTS	O
GUIDLINES	PRINCIPLES	
	MODELS	O
PRACTICES	METHODOLOGIES	L

LANGUAGE DEVELOPMENT

* Requirements completed before development

* Competitive procurement used for design

* Formal planned test and evaluation phase

* Massive public commentary used

* Design team used

* Strict standardization control

SOFTWARE ENGINEERING

* Goals of Software Engineering
* Principles of Software Engineering

GOALS OF SOFTWARE ENGINEERING

- ★ MODIFIABILITY
- ★ RELIABILITY
- ★ EFFICIENCY
- ★ UNDERSTANDABILITY

PRINCIPLES OF SOFTWARE ENGINEERING

- Abstraction
- Modularity
- Localization
- Information hiding
- Completeness
- Confirmability
- Uniformity

ABSTRACTION

* The process of separating out the important parts of something while ignoring the inessential details

* Separates the "what" from the "how"

* Reduces the level of complexity

* There are levels of abstraction within a system

MODULARITY

* Purposeful structuring of a system into parts which work together

* Each part performs some smaller task of the overall system

* Can concentrate and develop parts independently as long as interfaces are defined and shared

* Can develop hierarchies of management and implementation

LOCALIZATION

* Putting things that logically belong together in the same physical place

INFORMATION HIDING

* Puts a wall around localized details

* Prevents reliance upon details and causes focus of attention to interfaces and logical properties

COMPLETENESS

* Ensuring all important parts are present
* Nothing left out

CONFIRMABILITY

* Developing parts that can be effectively tested

UNIFORMITY

* No unnecessary differences across a system

FEATURES OF Ada

* Supports Large System Development
* Supports Structured Programming
* Supports Top-Down Development
* Supports Strong Data Typing
* Supports Data Abstraction
* Supports Information Hiding and Data Encapsulation

SYSTEMS ENGINEERING

* Analyze problem

* Break into solvable parts

* Implement parts

* Test parts

* Integrate parts to form total system

* Test total system

REQUIREMENTS FOR EFFECTIVE SYSTEMS ENGINEERING

* Ability to express architecture

* Ability to define and enforce interfaces

* Ability to create independent components

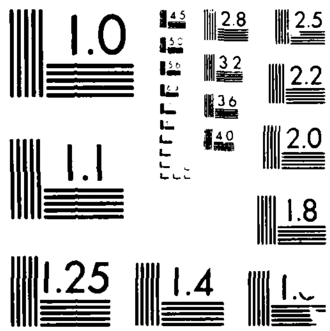
* Ability to separate architecture issues from implementation issues

Overview of Important Ada Features

- Readability
- Program Units
- Separate Compilation
- Subprograms
- Packages
- Strong Typing
- Typing Structures
- Data Abstraction
- Tasks
- Exceptions
- Generics
- Low Level Features

AD-A192 874 ASEET ADVANCED ADA WORKSHOP JANUARY 1988(U) ADA JOINT PROGRAM OFFICE ARLINGTON VA JAN 88 2/5

UNCLASSIFIED F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

READABILITY

* Ada was engineered with the understanding that programming is a human activity

* Features are provided that allow a maintenance person to quickly grasp the meaning of a particular program and to understand its structure

* Readability is more than just a language issue

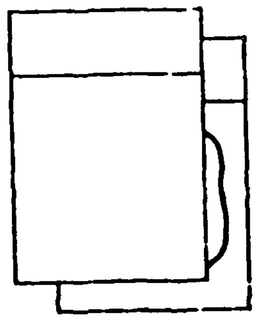
PROGRAM UNITS

* Components of Ada which together form a working Ada software system

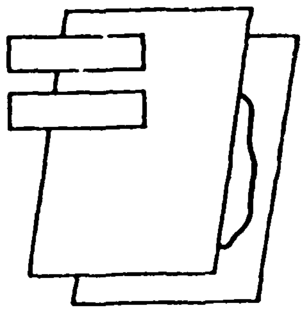
* Express the architecture of a system

* Define and enforce interfaces

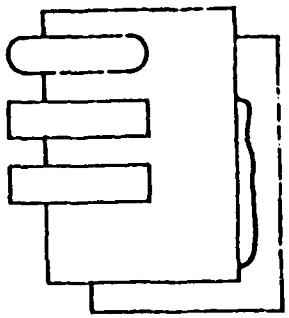
PROGRAM UNITS



SUBPROGRAMS — Working components that perform some action



TASKS — Performs actions in parallel with other program units

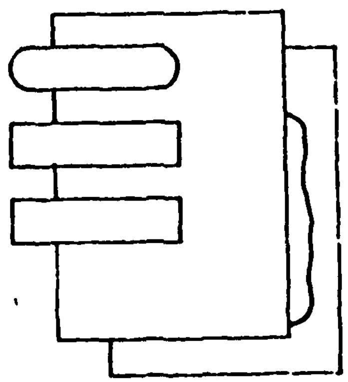


PACKAGES — A mechanism for collecting entities together into logical units

PROGRAM UNITS

* Consist of two parts: specification and body

SPECIFICATION: Defines the interface between the program unit and other program units (the **WHAT**)



BODY: Defines the implementation of the program unit (the **HOW**)

PROGRAM UNITS

* The specification of the program unit is the only means of connecting program units

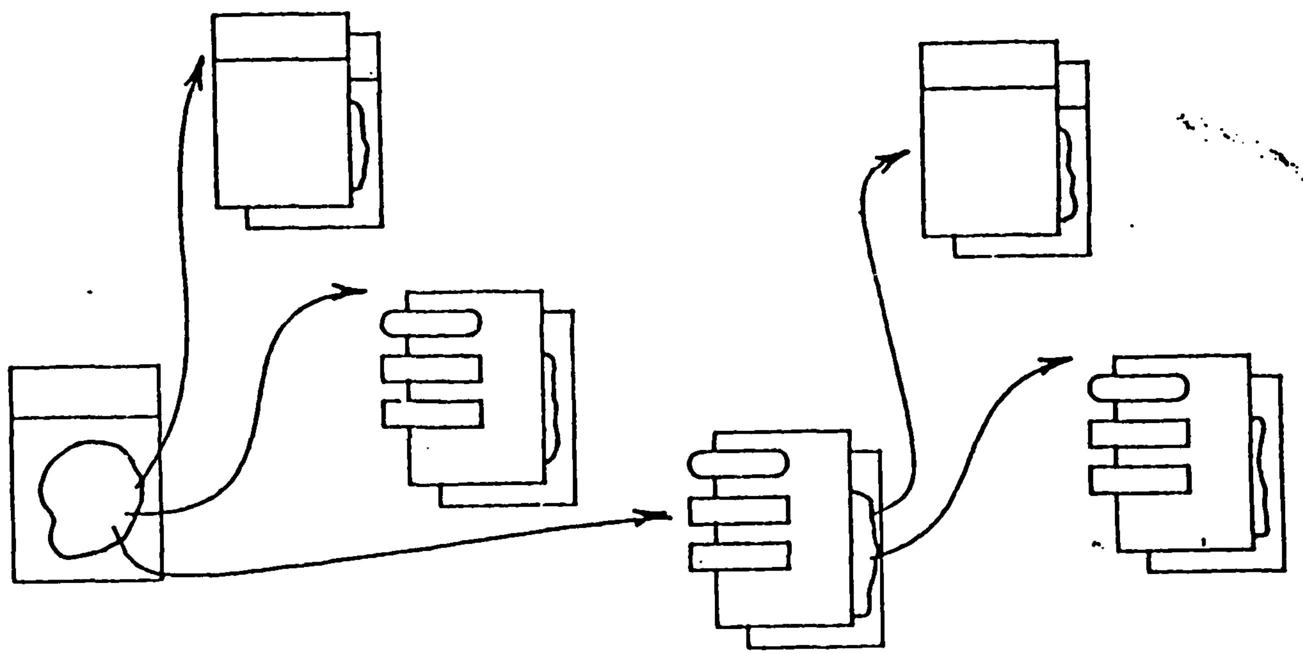
* The interface is enforced

* The body of a program unit is not accessible to other program units

* There is a clear distinction between architecture and implementation

SEPARATE COMPILATION

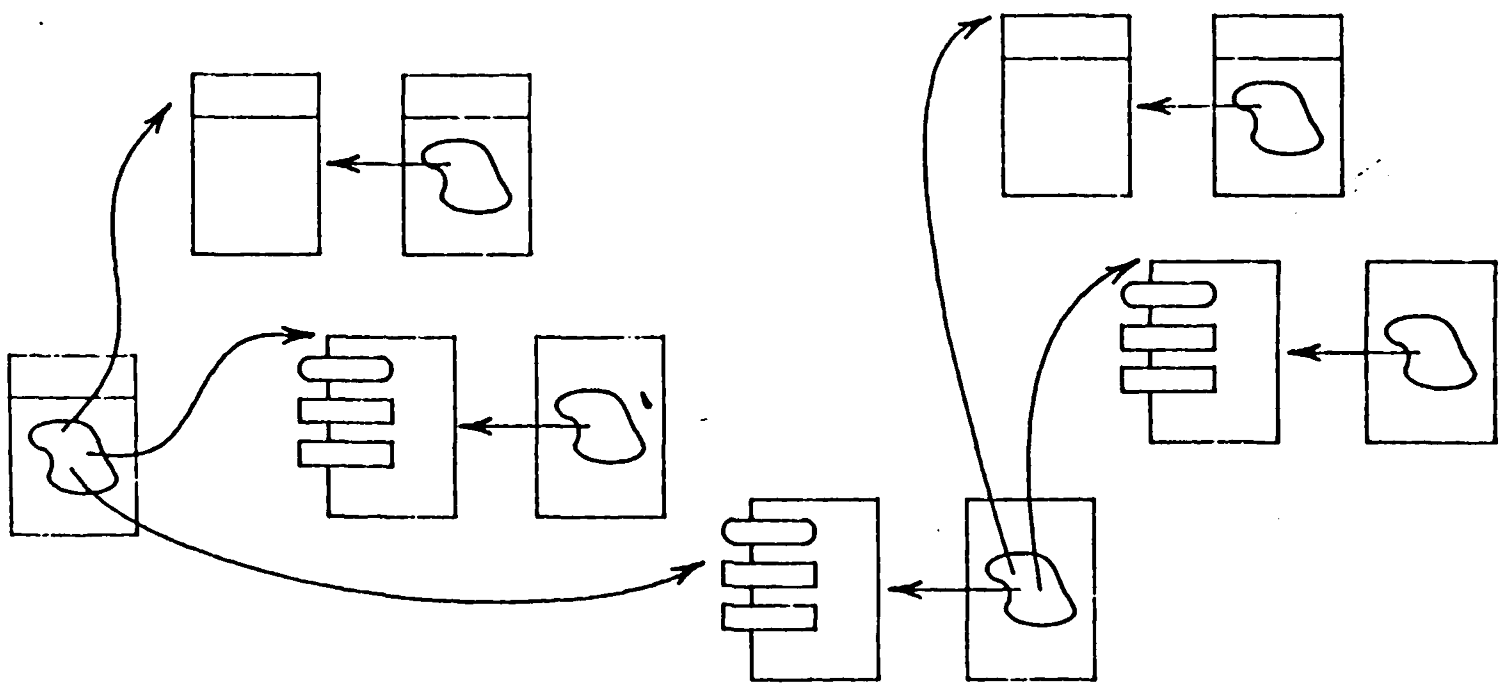
* Program units may be separately compiled
* Separate compilation is possible because of the separation of specification and body
* A system is put together by referencing the specifications of other program units



SEPARATE COMPILATION

* A program unit's specification may be compiled separately from its body

* Realizes not only a logical distinction between architecture and implementation, but also a physical distinction



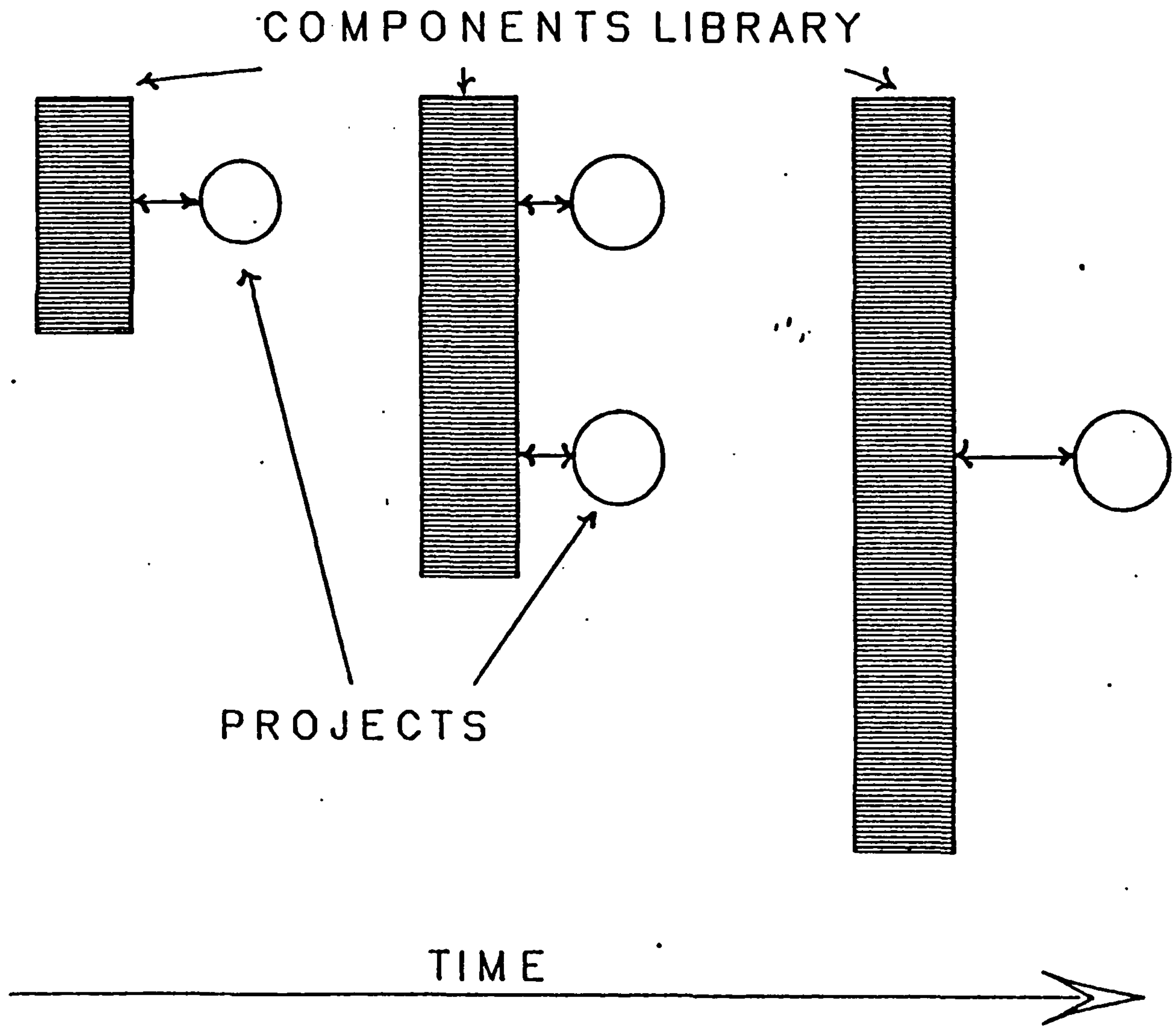
SEPARATE COMPILATION

* Allows development of independent software components

* Currently we all but lose the human effort going into software; it is disposable

* Separate compilation allows us to reuse components and keep our investment

SOFTWARE COMPONENTS



DISCRETE COMPONENTS

* Allow a system to be composed of black boxes

* Provide clear, understandable functions

* Black boxes can be more effectively validated and verified

* Prevalent across engineering disciplines

SUBPROGRAMS

* A program unit that performs a particular action

    – Procedures
    – Functions

* Contains an interface (parameter part) mechanism to pass data to and from the subprogram

* The basic discrete component which acts like a black box

* Gives ability to express abstract actions

MAJOR FEATURES OF Ada

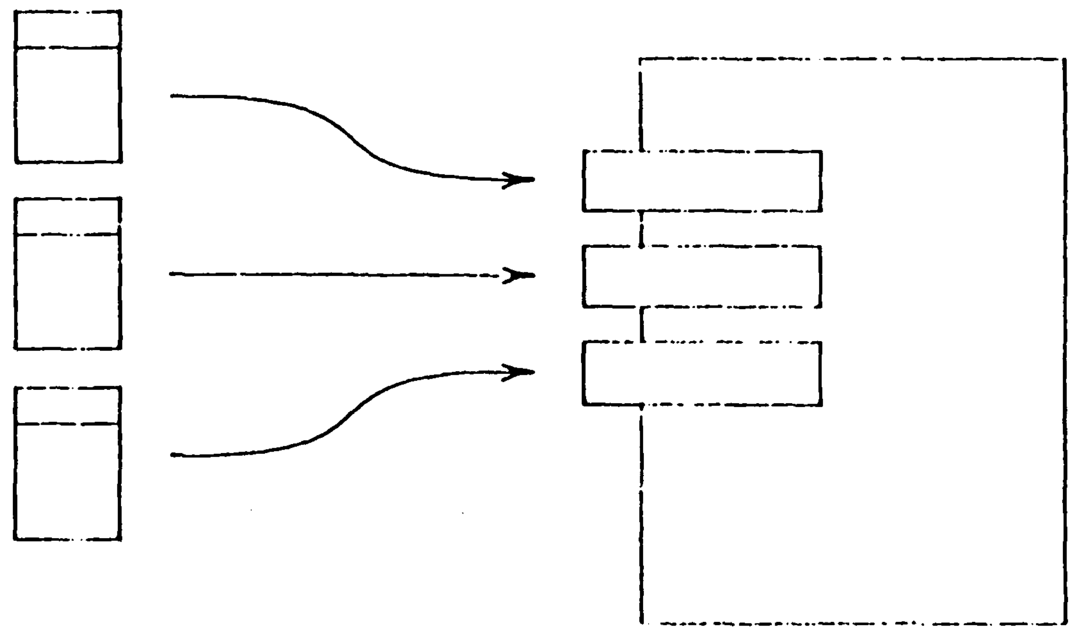
* Packages
* Strong Typing
* Typing Structures
* Data Abstraction
* Tasks
* Exceptions
* Generics

PACKAGES

* Definition
* Components of a Package
  -- Specification
  -- Body
* Goals and Principles of Software Engineering Supported

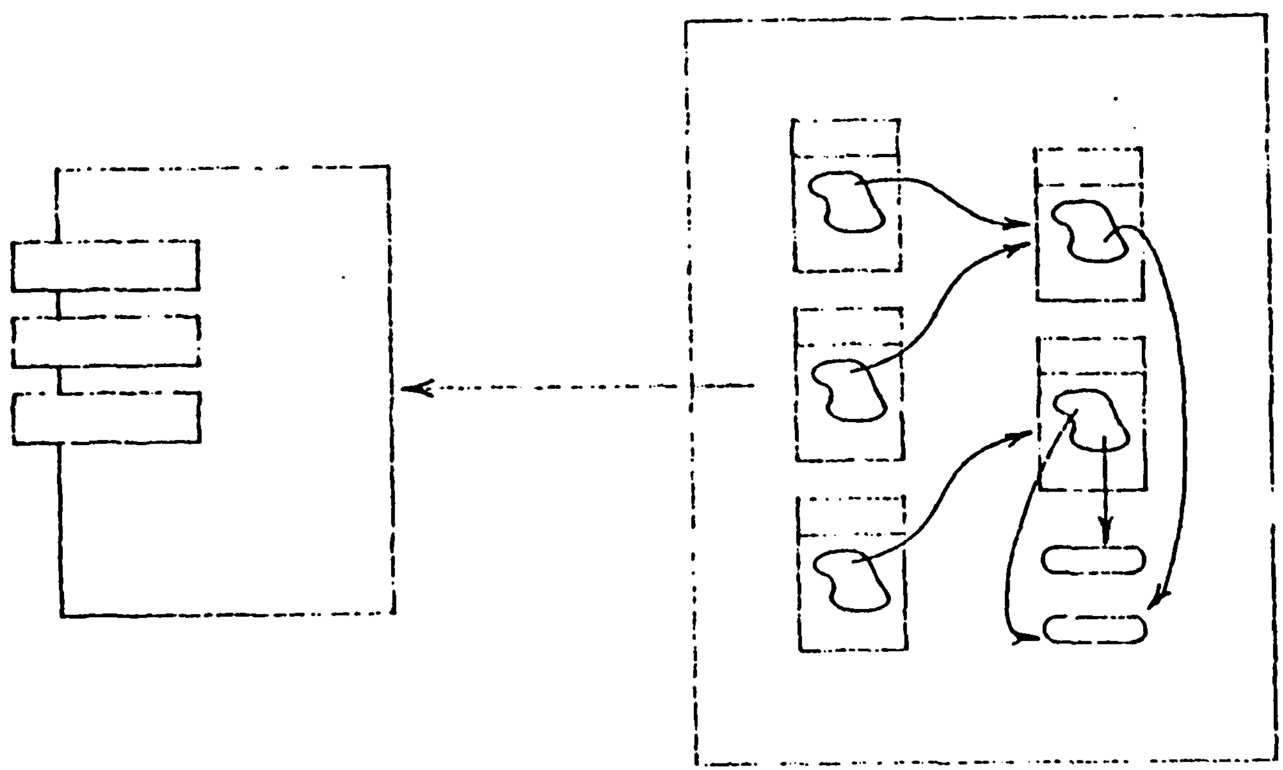
PACKAGES

* Program units that allow us to collect logically related entities in one physical place
* Allow the definition of reusable software components/resources
* A fundamental feature of Ada which allow a change of mindset
* An architecture–oriented feature



PACKAGES

* Place a "wall" around resources
* Export resources to users of a package
* May contain local resources hidden from the user of a package



Program Units

```
package ROBOT_CONTROL is

   type SPEED is range 0..100;
   type DISTANCE is range 0..500;
   type DEGREES is range 0..359;
   procedure GO_FORWARD ( HOW_FAST : in SPEED;
                          HOW_FAR : in DISTANCE );
   procedure REVERSE ( HOW_FAST : in SPEED;
                       HOW_FAR : in DISTANCE );
   procedure TURN ( HOW_MUCH : in DEGREES );

end ROBOT_CONTROL;
```

```
with ROBOT_CONTROL;

procedure DO_A_SQUARE is
begin

    ROBOT_CONTROL.GO_FORWARD( HOW_FAST => 100,
                              HOW_FAR => 20 );
    ROBOT_CONTROL.TURN( 90 );
    ROBOT_CONTROL.GO_FORWARD( 100, 20 );
    ROBOT_CONTROL.TURN( 90 );
    ROBOT_CONTROL.GO_FORWARD( 100, 20 );
    ROBOT_CONTROL.TURN( 90 );
    ROBOT_CONTROL.GO_FORWARD( 100, 20 );
    ROBOT_CONTROL.TURN ( 90 );

end DO_A_SQUARE;
```

Program Units

Package bodies

```
--Define local declarations
--Define implementation of subprograms
-- defined in specification
package body ROBOT_CONTROL is
    --local declarations
    procedure RESET_SYSTEM is
    begin
        --implementation
    end RESET_SYSTEM;
    procedure GO_FORWARD...is...
    procedure REVERSE...is...
    procedure TURN...is...
end ROBOT_CONTROL;
```

PACKAGES

DIRECTLY SUPPORT:

* Abstraction
* Information hiding
* Modularity
* Localization

* Understandability
* Efficiency
* Reliability and safety
* Modifiability
* Correctness

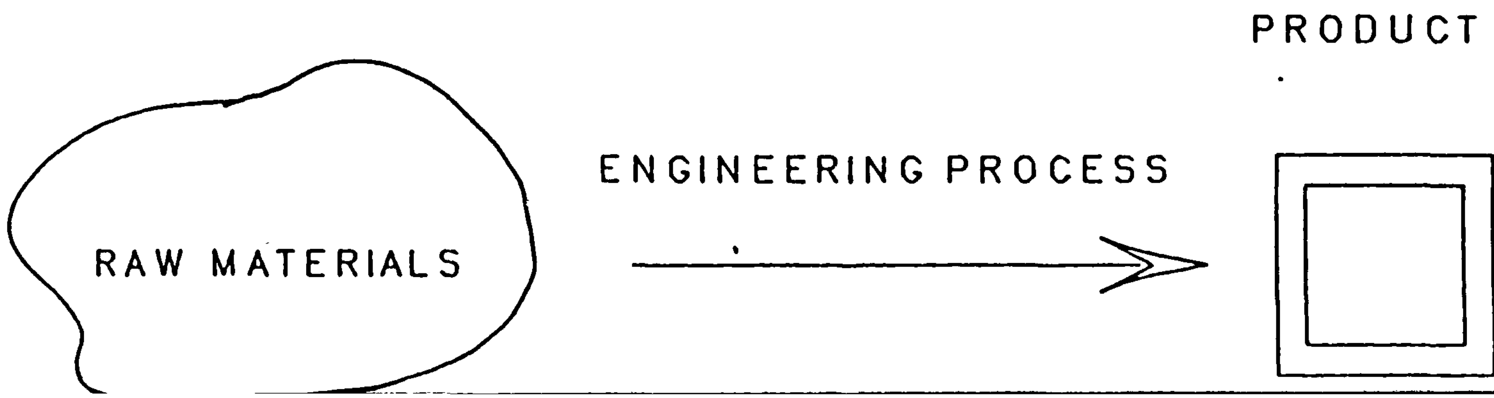
STRONG TYPING

* Raw Materials for Software Engineering
* Effects of Strong Typing
* Goals and Principles of Software Engineering Supported

THE RAW MATERIALS OF ENGINEERING

* All engineering disciplines shape raw materials into a finished product

* The materials and methods combine to define different disciplines



STRUCTURING RAW MATERIALS

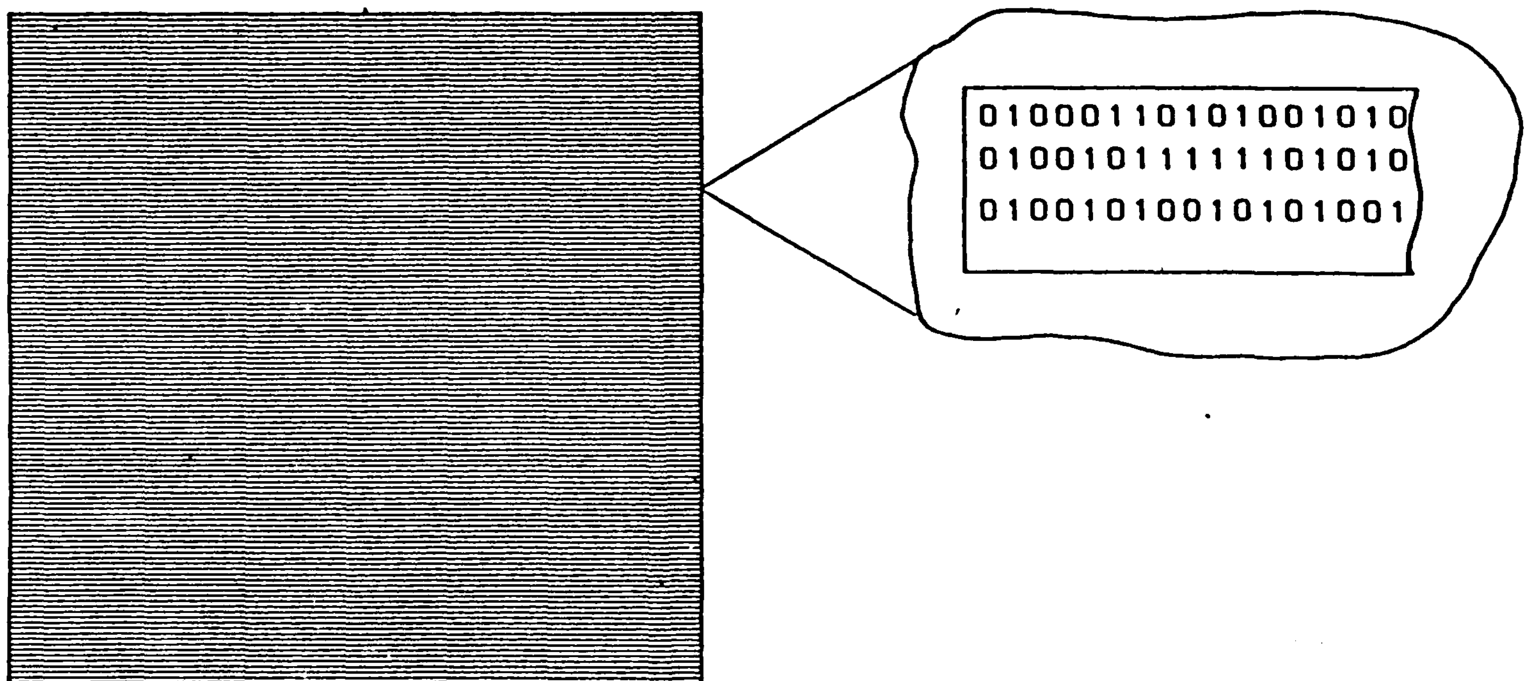
* There is a requirement to structure raw materials

  - To quantify
  - To manage
  - To test
  - To validate

* Methods of structuring vary across disciplines

SOME RAW MATERIALS OF SOFTWARE ENGINEERING

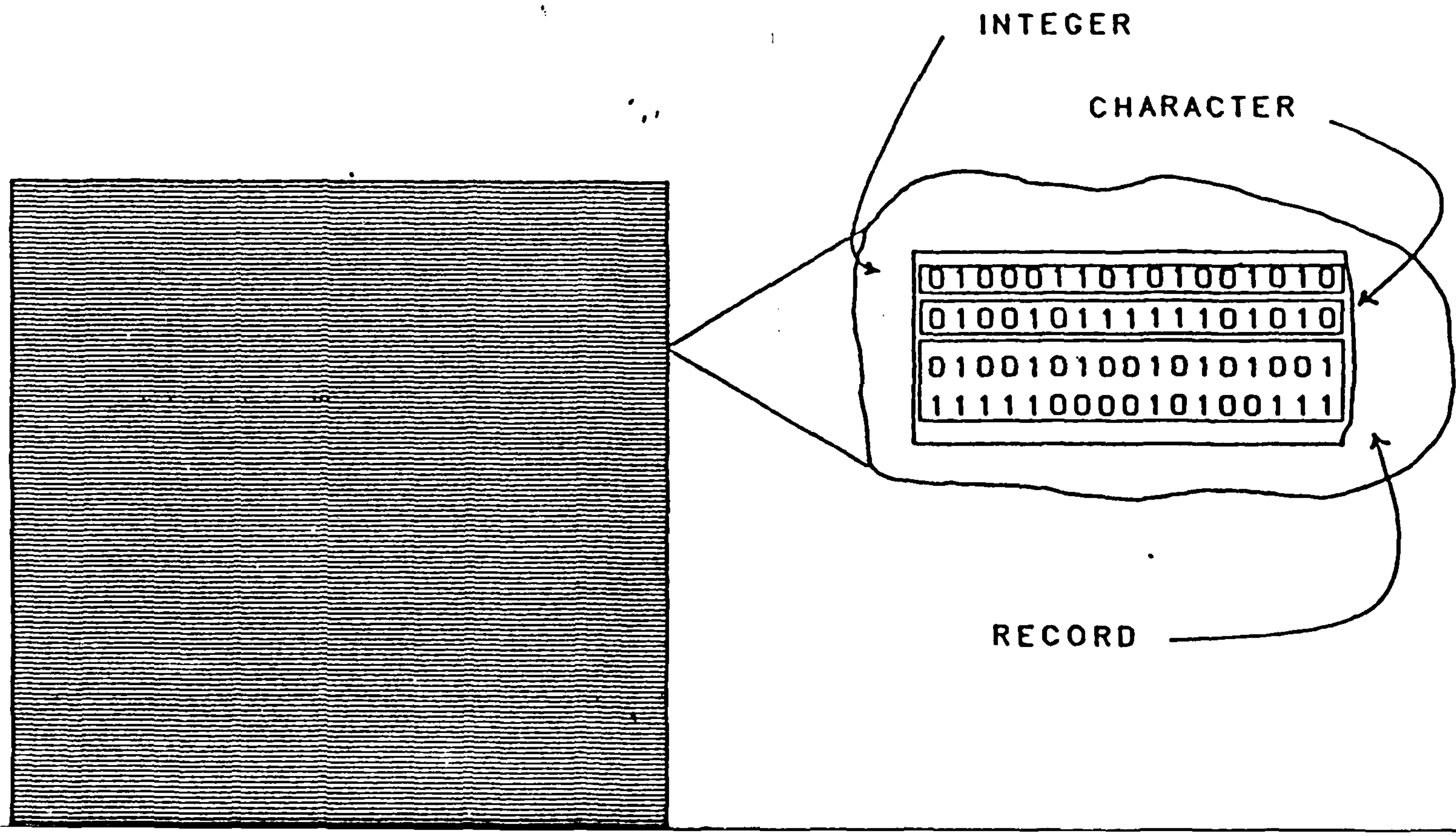
* Binary switches
* Computer memory locations
* Data



STRONG TYPING

* Defines structure of data (mapping)

* Enforces structure of data



STRONG TYPING

* Enforces abstraction of structure on data

* Increases confidence of correctness

* Increases reliability and safety

* Promotes understandability and maintainability

Types

--A type consists of a set of values that objects of the type may take on, and a set of operations applicable to those values

--Ada is a strongly typed language!

*Every object must be declared of some type name
*Different type names may not be implicitly mixed
*Operations on a type must preserve the type

```
AN_INTEGER     : INTEGER;
A_FLOAT_NUMBER : FLOAT ;
ANOTHER_FLOAT  : FLOAT;

A_FLOAT_NUMBER := ANOTHER_FLOAT + AN_INTEGER;
        --illegal
```

TYPING STRUCTURES

* Discrete Data Types
  -- Enumeration
  -- Integer
* Real Data Types
  -- Fixed Point (Absolute Error)
  -- Floating Point (Relative Error)
* Composite Types
  -- Arrays (Homogeneous)
  -- Records (Heterogeneous)
* Dynamic Types
  -- Access Types
* Abstract Data Types
  -- Private
  -- Limited Private

TYPING STRUCTURES

* Variety of problems requires a variety of structuring capabilities

* Ada provides a rich variety or types

TYPING STRUCTURES IN Ada

* Discrete data
  - Enumeration
  - Integer

* Real data
  - Fixed point (absolute error)
  - Floating point (relative error)

* Composite data
  - Arrays (homogeneous)
  - Records (heterogeneous)

* Dynamic data
  - Access types

Types

Integers

```
--Define a set of exact, consecutive values
USER DEFINED
          type ALTITUDE is range 0..100_000;
          type DEPTH is range 0..20_000;
          PLANES_HEIGHT : ALTITUDE;
          DIVER_DEPTH : DEPTH;

  begin
     PLANES_HEIGHT := 10_000;
     PLANES_HEIGHT := 200_000;  -- error
     PLANES_HEIGHT := DIVER_DEPTH; -- error
  end;
```

Types

Enumeration

--Define a set of ordered enumeration values
--Used in array indexing, case statements,
--   and looping

USER DEFINED

```
type SUIT is (CLUBS, HEARTS, DIAMONDS, SPADES);
type COLOR is (RED, WHITE, BLUE);
type SWITCH is (OFF, ON);
type EVEN DIGITS is ('2','4','6','8');
type MIXED is (ONE,'2',THREE,'*','!',more);
```

where CLUBS < HEARTS < DIAMONDS < SPADES
(pos 0) (pos 1) (pos 2) (pos 3)

Types

Fixed point types

-- Absolute bound on error
-- Larger error for smaller numbers (around zero)

USER DEFINED

```
type INCREMENT is delta 1.0/8 range 0.0 .. 1.0;
```

0, 1*2e-3, 2*2e-3, 4*2e-3, 5*2e-3,...

PREDEFINED

DURATION --> (Used for "delay" statements)

Types

Floating point types

-- Relative bound of error
-- Defined in terms of significant digits
-- More accurate at smaller numbers, less at larger

USER DEFINED

type NUMBERS is digits 3 range 0.0 .. 20_000;

0.001, 0.002, 0.003...999.0,1000.0,1001.0...,10000.0,10100.0

PREDEFINED

FLOAT

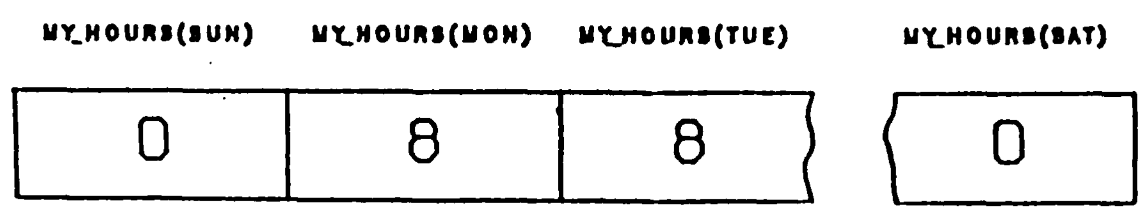
Types

Arrays — constrained / unconstrained

CONSTRAINED

-- Indices are static for all objects of that type

```
type HOURS is range 0..40;
type DAYS is ( SUN,MON,TUE,WED,THU,FRI,SAT );
type WORK_HOURS is array( DAYS ) of HOURS;

MY_HOURS : WORK_HOURS := ( 0,8,8,7,6,1,0 );
```

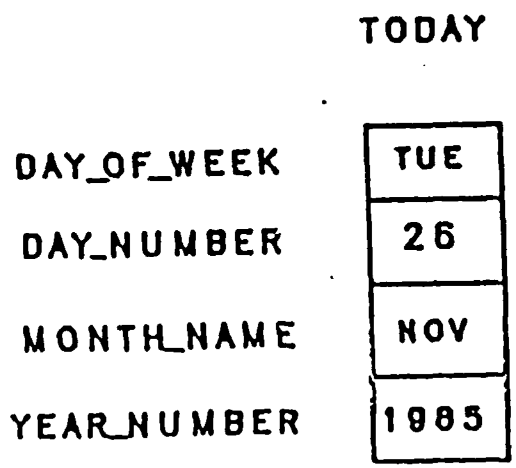


Types

Records

undiscriminated
discriminated
variant

```
UNDISCRIMINATED
   type DAYS is ( MON,TUE,WED,THU,FRI,SAT,SUN );
   type DAY is range 1..31;
   type MONTH is (JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,
                  SEP,OCT,NOV,DEC);
   type YEAR is range 0..2085;
   type DATE is record
      DAY_OF_WEEK : DAYS;
      DAY_NUMBER : DAY;
      MONTH_NAME : MONTH;
      YEAR_NUMBER : YEAR;
   end record;
   TODAY : DATE;
 begin
   TODAY.DAY_OF_WEEK := TUE;
   TODAY.DAY_NUMBER := 26;
   TODAY.MONTH_NAME := NOV;
```



DATA ABSTRACTION

* Definition

* Goals and Principles of Software Engineering Supported

* Baskin-Robbins Ice Cream Example

DATA ABSTRACTION

* Combines primitive raw materials to form higher level structures
* Levels of abstraction
* Enforces an abstraction on a higher level structure
* Prohibits use of implementation details
* Promotes understandability
* Promotes modifiability

DATA ABSTRACTION AND PRIVATE TYPES

* Private types directly implement data abstraction

* Directly implement information hiding

```
package B_R is

    type NUMBERS is range 0 ..99;

    procedure TAKE ( A_NUMBER : out NUMBERS );

    function NOW_SERVING return NUMBERS;

    procedure SERVE ( NUMBER : NUMBERS );

end B_R;
```

```
with B_R; use B_R;
procedure ICE_CREAM is
   YOUR_NUMBER : NUMBERS;
begin
   TAKE ( YOUR_NUMBER );
   loop
      if NOW_SERVING = YOUR_NUMBER then
         SERVE ( YOUR_NUMBER );
         exit;
      end if;
   end loop;
end ICE_CREAM;
```

```
with B_R; use B_R;
procedure ICE_CREAM is

  YOUR_NUMBER : NUMBERS;

begin

  TAKE ( YOUR_NUMBER );
  loop
    if NOW_SERVING = YOUR_NUMBER then
      SERVE ( YOUR_NUMBER );
      exit;
    else
     YOUR_NUMBER := YOUR_NUMBER - 1;
    end if;
  end loop;

end ICE_CREAM;
```

```
package B_R is

   type NUMBERS is private;

   procedure TAKE ( A_NUMBER : out NUMBERS );
   function NOW_SERVING return NUMBERS;
   procedure SERVE ( NUMBER : in NUMBERS );

private

   type NUMBERS is range 0..99;

end B_R;
```

```
with B_R; use B_R;
procedure ICE_CREAM is

   YOUR_NUMBER : NUMBERS;

begin

   TAKE ( YOUR_NUMBER );
   loop

      if NOW_SERVING = YOUR_NUMBER then
         SERVE ( YOUR_NUMBER );
         exit;
      else
         YOUR_NUMBER := NOW_SERVING;
      end if;
   end loop;

end ICE_CREAM;
```

```
package B_R is

   type NUMBERS is limited private;

   procedure TAKE ( A_NUMBER : out NUMBERS );
   function NOW_SERVING return NUMBERS;
   procedure SERVE ( NUMBER : in NUMBERS );
   function "=" ( LEFT, RIGHT : in NUMBERS ) return
                                            BOOLEAN;

private

   type NUMBERS is range 0..99;

end B_R;
```

```
with B_R; use B_R;
procedure ICE_CREAM is

   YOUR_NUMBER : NUMBERS;
   procedure GO_TO_DQ is separate;

 begin
   TAKE ( YOUR_NUMBER );
   loop
     if NOW_SERVING = YOUR_NUMBER then
        SERVE ( YOUR_NUMBER );
        exit;
     else
        GO_TO_DQ;
        exit;
     end if;
   end loop;

 end ICE_CREAM;
```

TASKS

* Definition

* Goals and Principles of Software Engineering Supported

* Example

TASKS

* Program unit that acts in parallel with other entities

* Directly implements those parts of embedded systems which act in parallel

* Takes advantage of move toward parallel hardware architectures

  - Fault tolerance
  - Distributed systems

* Eliminates need to introduce additional complexity into a system

Tasks

```
procedure SENSOR_CONTROLLER is

    function OUT_OF_LIMITS return BOOLEAN;
    procedure SOUND_ALARM;


    task MONITOR_SENSOR; -- specification
    task body MONITOR_SENSOR is  -- body
    begin
          loop
             if OUT_OF_LIMITS then
                SOUND_ALARM;
             end if;
          end loop;
    end MONITOR_SENSOR;

    function OUT_OF_LIMITS return BOOLEAN is separate;
    procedure SOUND_ALARM is separate;
begin
   null;  -- Task is activated here
end SENSOR_CONTROLLER;
```

Tasks

```
-- a basic task with no communication
 with TEXT_IO; use TEXT_IO;
 procedure COUNT_NUMBERS is
       package INT_IO is new INTEGER_IO (INTEGER);
       use INT_IO;
       task COUNT_SMALL;
       task COUNT_LARGE;

       task body COUNT_SMALL is
       begin
            for INDEX in -100..0 loop
                 PUT(INDEX);
                 NEW_LINE;
             end loop;
        end COUNT_SMALL;

        task body COUNT_LARGE is
        begin
             for INDEX in 0..100 loop
                  PUT(INDEX);
                  NEW_LINE;
               end loop;
           end COUNT_LARGE;

    begin
        null; --tasks are started here
    end COUNT_NUMBERS;
```

EXCEPTIONS

* Definition

* Goals and Principles of Software Engineering Supported

* Types of Exceptions in Ada

  -- Pre-defined Exceptions

  -- User-defined Exceptions

* Example

SOFTWARE RELIABILITY AND SAFETY

* Errors will occur
    - Hardware
    - Software
* Real time systems must be able to operate in a degraded mode
* Reliablity and safety must be engineered into a system
* Traditional languages lack specific features for dealing with errors and exceptional situations

EXCEPTIONS

* Deal specifically with errors and exceptional situations

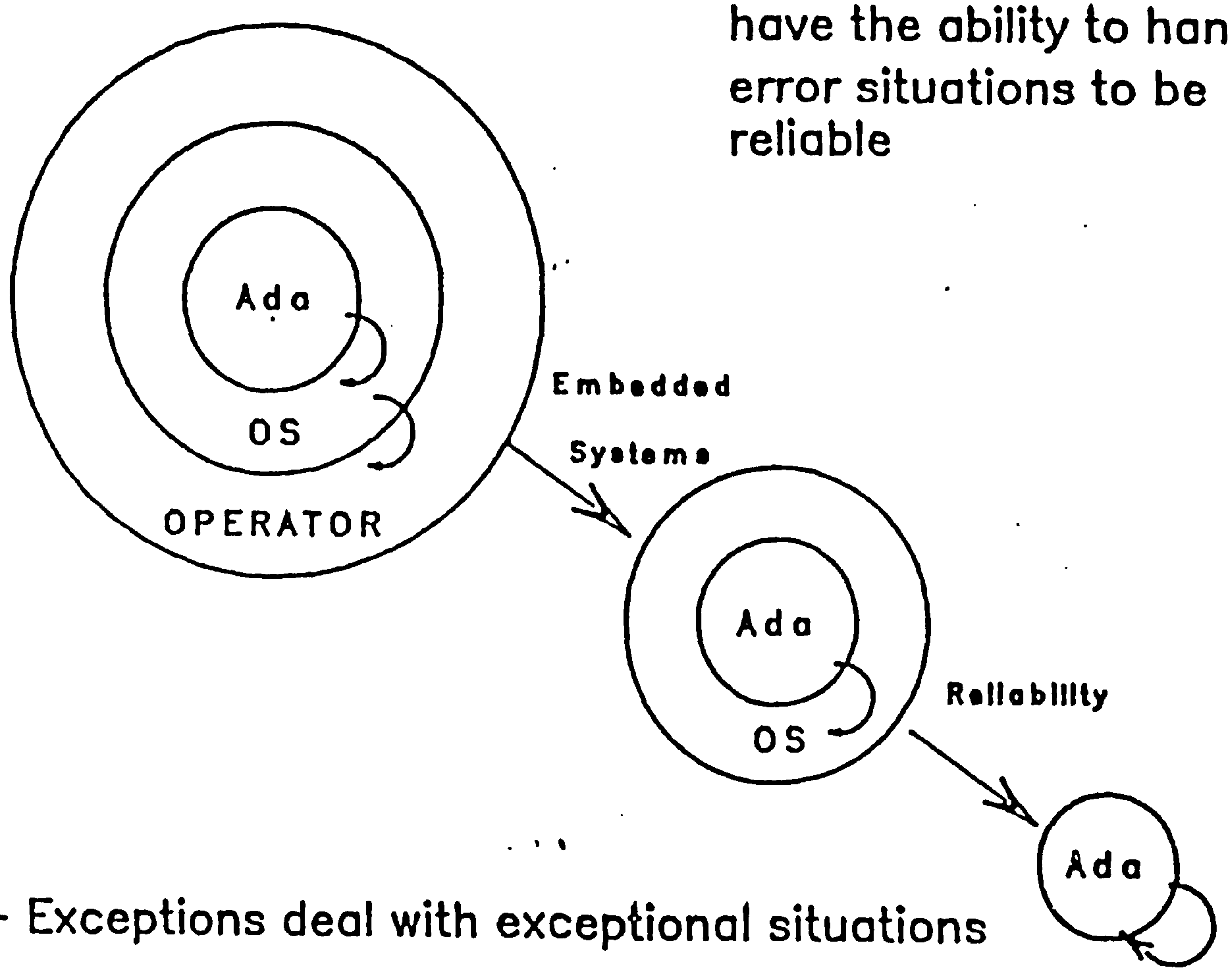
* When an exception is raised processing is suspended and control is passed to an appropriate exception handler
  - Try again
  - Fix error
  - Propogate exception

* Increase reliability

* Reduce complexity

Exceptions

-- Real time systems must have the ability to handle error situations to be reliable



-- Exceptions deal with exceptional situations

Exceptions

-- When an exception situation occurs, the exception is said to be "raised"

-- What happens then, depends on the presence or absence of an exception handler

```
begin
  loop
     GET ( A_NUMBER );
     NEW_LINE;
     PUT("The number is");
     PUT ( A_NUMBER );
     NEW_LINE;
  end loop;
end GET_NUMBERS;
```

Exceptions

```
begin
  loop
    begin
      GET ( A_NUMBER );
      NEW_LINE;
      PUT ( "The number is ");
      PUT ( A_NUMBER );
      NEW_LINE;
    exception
      when DATA_ERROR => PUT_LINE("Bad number, try again");
    end;
  end loop;
end GET_NUMBERS;.
```

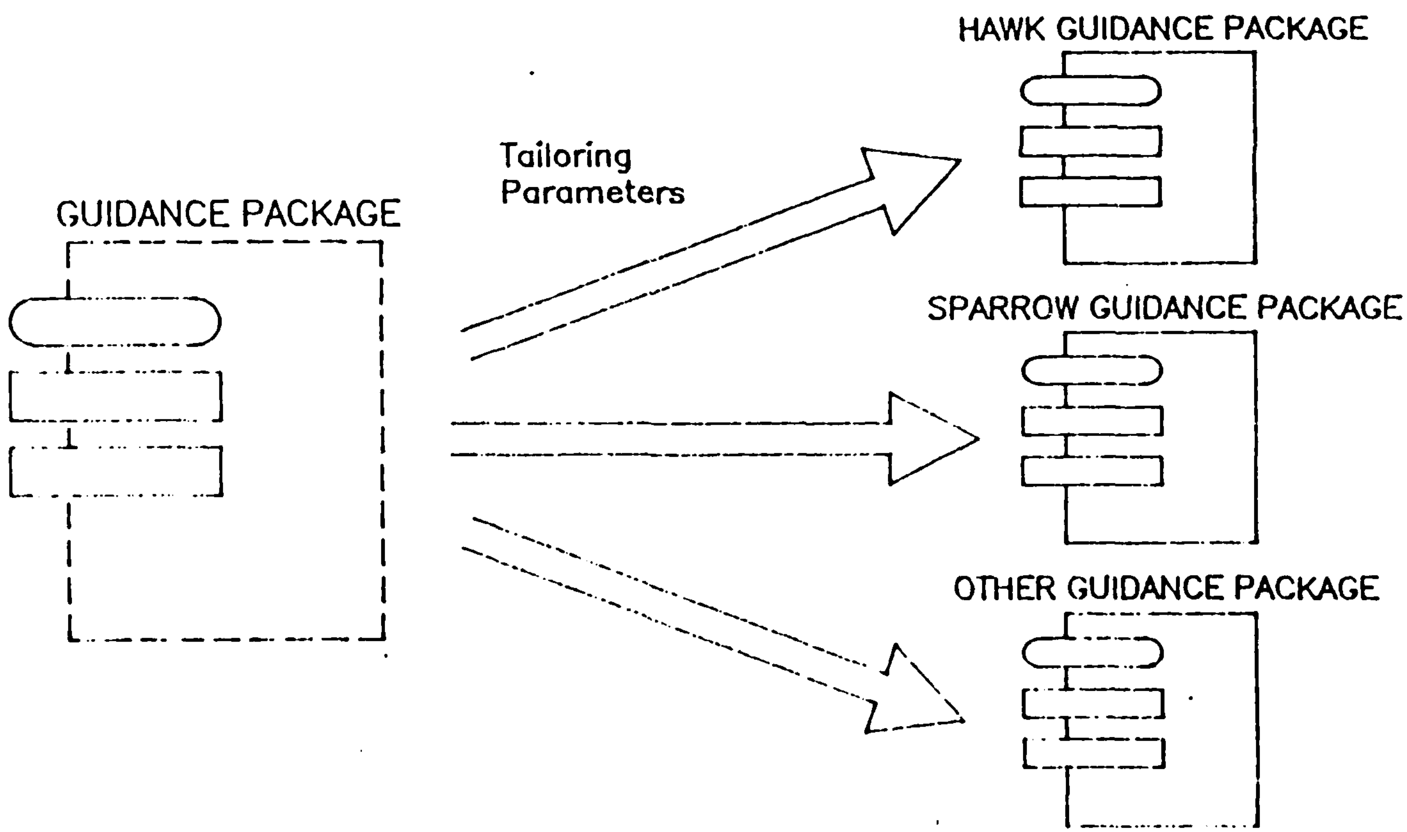
GENERICS

* Definition
* Goals and Principles of Software Engineering Supported
* Example of Generic Unit Use

GENERICS

* A generic is a tailorable template for a program unit

* Increases reusable software component capability by an order of magnitude



GENERICS

* Reduce size of program text

* Reduce need to reinvent the wheel

* Increase reliability by allowing reuse of known reliable components

Generics

```
procedure INTEGER_SWAP (FIRST_INTEGER, SECOND_INTEGER:
                        in out INTEGER) is

        TEMP : INTEGER;

begin

        TEMP           := FIRST_INTEGER;
        FIRST_INTEGER  := SECOND_INTEGER;
        SECOND_INTEGER := TEMP;

end INTEGER_SWAP;
```

Generics

```
generic
    type ELEMENT is private;
procedure SWAP (ITEM_1;ITEM_2:in out ELEMENT);

procedure SWAP(ITEM_1,ITEM_2:in out ELEMENT) is
    TEMP:ELEMENT;
begin
    TEMP := ITEM_1;
    ITEM_1 := ITEM_2;
    ITEM_2 := TEMP;
end SWAP;
```

Generics

```
with SWAP;
procedure EXAMPLE is
  procedure INTEGER_SWAP is new SWAP(INTEGER);
  procedure CHARACTER_SWAP is new SWAP(CHARACTER);
  NUM_1, NUM_2 : INTEGER;
  CHAR_1, CHAR_2 : CHARACTER;
begin
  NUM_1 := 10;
  NUM_2 := 25;
  INTEGER_SWAP(NUM_1, NUM_2 );
  CHAR_1 := 'A';
  CHAR_2 := 'S';
  CHARACTER_SWAP(CHAR_1, CHAR_2);
end EXAMPLE;
```

SUMMARY

* Basic Problem
    - Projection to the 1990's
    - A Macro Solution
* A Practical Solution
    - Software Engineering
    - Ada
* Software Engineering
    - Goals
    - Principles
* Why Ada ?
    - Features of Ada
    - Software Engineering Applications

PACKAGES

DIRECTLY SUPPORTS:

* ABSTRACTION
* MODULARITY
* LOCALIZATION
* INFORMATION HIDING
  UNIFORMITY
* COMPLETENESS
* CONFIRMABILITY

* MODIFIABILITY
* RELIABILITY
* EFFICIENCY
* UNDERSTANDABILITY

TYPING

DIRECTLY SUPPORTS:

ABSTRACTION
MODULARITY
LOCALIZATION
INFORMATION HIDING
UNIFORMITY

* COMPLETENESS
* CONFIRMABILITY

* MODIFIABILITY
* RELIABILITY
* EFFICIENCY
* UNDERSTANDABILITY

DATA ABSTRACTION

DIRECTLY SUPPORTS:

* ABSTRACTION
  MODULARITY
  LOCALIZATION
* INFORMATION HIDING
  UNIFORMITY
* COMPLETENESS
  CONFIRMABILITY

* MODIFIABILITY
* RELIABILITY
  EFFICIENCY
* UNDERSTANDABILITY

EXCEPTIONS

DIRECTLY SUPPORTS:

ABSTRACTION
* MODULARITY
* LOCALIZATION
INFORMATION HIDING
* UNIFORMITY
* COMPLETENESS
* CONFIRMABILITY

MODIFIABILITY
* RELIABILITY
* EFFICIENCY
* UNDERSTANDABILITY

TASKS

DIRECTLY SUPPORTS:

* ABSTRACTION
* MODULARITY
LOCALIZATION
* INFORMATION HIDING
UNIFORMITY
* COMPLETENESS
* CONFIRMABILITY

MODIFIABILITY
* RELIABILITY
* EFFICIENCY
* UNDERSTANDABILITY

GENERICS

DIRECTLY SUPPORTS:

* ABSTRACTION
* MODULARITY
* LOCALIZATION
  INFORMATION HIDING
* UNIFORMITY
* COMPLETENESS
* CONFIRMABILITY

* MODIFIABILITY
* RELIABILITY
* EFFICIENCY
* UNDERSTANDABILITY

Tutorial on Ada® Exceptions

by
Major Patricia K. Lawlis

lawlis%asu@csnet-relay

Air Force Institute of Technology (AFIT)
and
Arizona State University (ASU)

13 January 1988

® Ada is a registered trademark of the U. S. Government - Ada Joint Program Office

References

- Student Handout, "Ada Applications Programmer - Advanced Ada Software Engineering", USAF Technical Training School, Keesler Air Force Base, July 1986.

- J. G. P. Barnes, Programming in Ada, Second edition, Addison-Wesley, 1984.

- Grady Booch, Software Engineering with Ada, Second Edition, Benjamin/Cummings, 1987.

- Putnam P. Texel, Introductory Ada: Packages for Programming, Wadsworth, 1986.

- Eugen N. Vasilescu, Ada Programming with Applications, Allyn and Bacon, 1986.

- ANSI/MIL-STD-1815A, "Military Standard - Ada Programming Language" (**LRM**), U. S. Department of Defense, 22 January 1983.

Outline

=> **Overview**

- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions
- Turning off exception checking
- Tasking exceptions
- More examples
- Summary

Overview

- What is an exception
- Ada exceptions
- Comparison
    - the American way
    - using exceptions

What Is an Exception

- A run time error
- An unusual or unexpected condition
- A condition requiring special attention
- Other than normal processing
- An important feature for debugging
- A critical feature for operational software

Ada Exceptions

- An exception has a name
  - may be predefined
  - may be declared
- The exception is raised
  - may be raised implicitly by run time system
  - may be raised explicitly by **raise** statement
- The exception is handled
  - exception handler may be placed in any **frame***
  - exception propagates until handler is found
  - if no handler anywhere, process aborts

* executable part surrounded by begin - end

The American Way

```
package Stack_Package is

        type Stack_Type is limited private;

        procedure Push (Stack               : in out Stack_Type;
                        Element             : in     Element_Type;
                        Overflow_Flag       : out    BOOLEAN);
        ...

end Stack_Package;


with TEXT_IO;
with Stack_Package; use Stack_Package;
procedure Flag_Waving is
        ...
        Stack     : Stack_Type;
        Element : Element_Type;
        Flag      : BOOLEAN;
begin
        ...
        Push (Stack, Element, Flag);
        if Flag then
                TEXT_IO.PUT ("Stack overflow");
                ...
        end if;
        ...
end Flag_Waving;
```

Using Exceptions

```
package Stack_Package is

        type Stack_Type is limited private;
        Stack_Overflow,
        Stack_Underflow : exception;

        procedure Push (Stack       : in out   Stack_Type;
                        Element     : in       Element_Type);
                  -- may raise Stack_Overflow
        ...
end Stack_Package;


with TEXT_IO;
with Stack_Package; use Stack_Package;
procedure More_Natural is
        ...
        Stack    : Stack_Type;
        Element : Element_Type;
begin
        ...
        Push (Stack, Element);
        ...
exception
        when Stack_Overflow =>
                TEXT_IO.PUT ("Stack overflow");
        ...
end More_Natural;
```

Outline

- Overview

=> **Naming an exception**

- Creating an exception handler
- Raising an exception
- Handling exceptions
- Turning off exception checking
- Tasking exceptions
- More examples
- Summary

Naming an Exception

- Predefined exceptions
- Declaring exceptions
- I/O exceptions

Predefined Exceptions

- In package STANDARD (also see chap 11 of LRM)

- CONSTRAINT_ERROR

  violation of range, index, or discriminant constraint...

- NUMERIC_ERROR

  execution of a predefined numeric operation cannot deliver a correct result

- PROGRAM_ERROR

  attempt to access a program unit which has not yet been elaborated...

- STORAGE_ERROR

  storage allocation is exceeded...

- TASKING_ERROR

  exception arising during intertask communication

Declaring Exceptions

exception_declaration ::= identifier_list : **exception**;

- Exception may be declared anywhere an object declaration is appropriate

- However, exception is not an object
  - may not be used as subprogram parameter, record or array component
  - has same scope as an object, but its effect may extend beyond its scope

Example:

```
procedure Calculation is

    Singular                    : exception;
    Overflow, Underflow         : exception;

begin
    ...
end Calculation;
```

I/O Exceptions

- Exceptions relating to file processing

- In predefined library unit IO_EXCEPTIONS
  (also see chap 14 of LRM)

- TEXT_IO, DIRECT_IO, and SEQUENTIAL_IO **with** it

```
package IO_EXCEPTIONS is

        NAME_ERROR    : exception;
        USE_ERROR     : exception;     --attempt to use
                                       --invalid operation
        STATUS_ERROR  : exception;
        MODE_ERROR    : exception;
        DEVICE_ERROR  : exception;
        END_ERROR     : exception;     --attempt to read
                                       --beyond end of file
        DATA_ERROR    : exception;     --attempt to input
                                       --wrong type
        LAYOUT_ERROR  : exception;     --for text processing

end IO_EXCEPTIONS;
```

Outline

- Overview

- Naming an exception

=> Creating an exception handler

- Raising an exception

- Handling exceptions

- Turning off exception checking

- Tasking exceptions

- More examples

- Summary

Creating an Exception Handler

- Defining an exception handler
- Restrictions
- Handler example

Defining an Exception Handler

- Exception condition is "caught" and "handled" by an exception handler

- Exception handler may appear at the end of any frame (block, subprogram, package or task body)

```
begin
    ...
exception
    -- exception handler(s)
end;
```

- Form similar to case statement

exception_handler ::=
    when exception_choice {| exception_choice} **=>**
        sequence_of_statements

exception_choice ::= exception_name | **others**

Restrictions

- Exception handlers must be at the end of a frame

- Nothing but exception handlers may lie between **exception** and **end** of frame

- A handler may name any visible exception declared or predefined

- A handler includes a sequence of statements

  - response to exception condition

- A handler for **others** may be used

  - must be the last handler in the frame

  - handles all exceptions not listed in previous handlers of the frame (including those not in scope of visibility)

  - can be the only handler in the frame

Handler Example

```
procedure Whatever is

    Problem_Condition : exception;

begin
    ...

exception

    when Problem_Condition =>
        Fix_It;

    when CONSTRAINT_ERROR =>
        Report_It;

    when others =>
        Punt;

end Whatever;
```

Outline

- Overview
- Naming an exception
- Creating an exception handler

=> **Raising an exception**

- Handling exceptions
- Turning off exception checking
- Tasking exceptions
- More examples
- Summary

Raising an Exception

- Elaboration and execution exceptions

- How exceptions are raised

- Effects of raising an exception

- Raising example

Elaboration and Execution Exceptions

- Elaboration exceptions occur when declarations are being elaborated
  - after a unit is "called"
  - before execution of the unit begins
  - can only be predefined exceptions
- Execution exceptions occur during execution of a frame
- Elaboration exceptions can also be considered as execution exceptions
  - depending on viewpoint
  - can consider as part of the execution of the last executable statement making the call to the unit being elaborated
  - this helps with understanding the consistency of the rules for exception handling

How Exceptions are Raised

- Implicitly by run time system

  - predefined exceptions

- Explicitly by **raise** statement

raise_statement ::= **raise** *[exception_name];*

  - the name of the exception must be visible at the point of the raise statement

  - a raise statement without an exception name is allowed only within an exception handler

Effects of Raising an Exception

(1) Control transfers to exception handler at end of frame being **executed** (if handler exists)

(2) Exception is lowered

(3) Sequence of statements in exception hander is executed

(4) Control passes to end of frame

- If frame does not contain an appropriate exception handler, the exception is propagated - effectively skipping steps 1 thru 3 and going straight to step 4

Raising Example

```
procedure Whatever is

        Problem_Condition       : exception;
        Real_Bad_Condition      : exception;

begin
        ...
        if Problem_Arises then
                raise Problem_Condition;                -- 1
        end if;
        ...
        if Serious_Problem then
                raise Real_Bad_Condition;               -- 1
        end if;
        ...
exception

        when Problem_Condition =>                       -- 2
                Fix_It;                                 -- 3

        when CONSTRAINT_ERROR =>                        -- 2
                Report_It;                              -- 3

        when others =>                                  -- 2
                Punt;                                   -- 3

end Whatever;                                           -- 4
```

Outline

- Overview
- Naming an exception
- Creating an exception handler
- Raising an exception

=> **Handling exceptions**

- Turning off exception checking
- Tasking exceptions
- More examples
- Summary

Handling Exceptions

- How exception handling can be useful
- Which exception handler is used
- Sequence of statements in exception handler
- Propagation
- Propagation example

How Exception Handling Can Be Useful

- Normal processing could continue if
  - cause of exception condition can be "repaired"
  - alternative approach can be used
  - operation can be retried
- Degraded processing could be better than termination
  - for example, safety-critical systems
- If termination is necessary, "clean-up" can be done first

Which Exception Handler Is Used

- When exception is raised, system looks for an exception handler at the end of the frame being executed

- If exception is raised during elaboration of the declarative part of a unit (unit is not yet ready to execute)

  - elaboration is abandoned and control goes to the end of the unit with the exception still raised

  - exception part of the unit is not searched for an appropriate handler

  - effectively, the calling unit will be searched for an appropriate handler
    - consistent with execution viewpoint

  - if elaboration of library unit, program execution is abandoned
    - all library units are elaborated with the main program

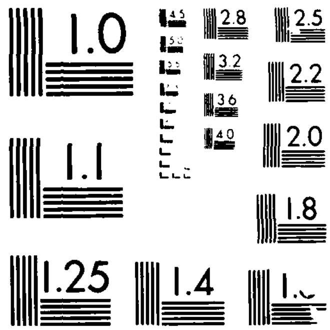
- If exception is raised in exception handler

  - handler may contain block(s) with handler(s)

  - if not handled locally within handler, control goes to end of frame with exception raised

AD-A192 074 ASEET ADVANCED ADA WORKSHOP JANUARY 1988(U) ADA JOINT PROGRAM OFFICE ARLINGTON VA JAN 88 3/5

UNCLASSIFIED F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Propagation

- Occurs if no handler exists in frame where execution exception is raised

- Always occurs if elaboration exception is raised

- Also occurs if **raise** statement is used in handler

- Exception is propagated dynamically

    - propagates from subprogram to unit calling it (not necessarily unit containing its declaration)

    - this can result in propagation outside its scope

    - task propagation follows same principle, but a little more complicated

- Propagation continues until

    - an appropriate handler is found

    - exception propagates to main program (still with no handler) and program execution is abandoned

Propagation Example

```
procedure Do_Nothing is
          --------------
     procedure Has_It is
          Some_Problem : exception;
     begin
          ...
          raise Some_Problem;
          ...
     exception
          when Some_Problem =>
               Clean_Up;
               raise;
     end Has_It;
          --------------
     procedure Calls_It is
     begin
          ...
          Has_It;
          ...
     end Calls_It;
          --------------
begin  -- Dc_Nothing
     ...
     Calls_It;
     ...
exception
     when others => Fix_Everything;
end Do_Nothing;
```

Outline

- Overview
- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions

=> **Turning off exception checking**

- Tasking exceptions
- More examples
- Summary

Turning Off Exception Checking

- Overhead vs efficiency
- Pragma SUPPRESS
- Check identifiers

Overhead vs Efficiency

- Exception checking imposes run time overhead

  - interactive applications will never notice
  - real-time applications have legitimate concerns but must not sacrifice system safety

- When efficiency counts

  - first, make program work (using good design)
  - be sure possible problems are covered by exception handlers
  - check if efficient enough - stop if it is

  - if not, study execution profile
    -- eliminate bottlenecks
    -- improve algorithm
    -- avoid "cute" tricks

  - check if efficient enough - stop if it is
  - if not, trade-offs may be necessary
  - some exception checks may be expendable since debugging is done

  - however, every suppressed check poses new possibilities for problems
    -- must re-examine possible problems
    -- must re-examine exception handlers

  - always keep in mind
    -- problems <u>will</u> happen
    -- critical applications must be able to deal with these problems

Moral

Improving the design is far better - and easier in the long run - than suppressing checks

Pragma SUPPRESS

- Only allowed immediately within a declarative part or immediately within a package specification

pragma SUPPRESS (identifier [,[**ON** =>] name]);

- identifier is that of the check to be omitted (next slide lists identifiers)

- name is that of an object, type, or unit for which the check is to be suppressed

-- if no name is given, it applies to the remaining declarative region

- An implementation is free to ignore the suppress directive for any check which may be impossible or too costly to suppress

Example:

```
pragma SUPPRESS (INDEX_CHECK, ON => Index);
```

Check Identifiers

- These identifiers are explained in more detail in chap 11 of the LRM

- Check identifiers for suppression of CONSTRAINT_ERROR checks

  ACCESS_CHECK
  DISCRIMINANT_CHECK
  INDEX_CHECK
  LENGTH_CHECK
  RANGE_CHECK

- Check identifiers for suppression of NUMERIC_ERROR checks

  DIVISION_CHECK
  OVERFLOW_CHECK

- Check identifier for suppression of PROGRAM_ERROR checks

  ELABORATION_CHECK

- Check identifier for suppression of STORAGE_ERROR check

  STORAGE_CHECK

Outline

- Overview
- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions
- Turning off exception checking

=> **Tasking exceptions**

- More examples
- Summary

Tasking Exceptions

- Exception handling is trickier for tasks
- Exceptions during task communication
- Tasking example

Exception Handling Is Trickier for Tasks

- Rules are not really different, just more involved
  - local exceptions handled the same within frames

If exception is raised

- during elaboration of task declarations
  - the exception TASKING_ERROR will be raised at the point of task activation (becomes execution exception in enclosing subprogram)
  - the task will be marked completed
- during execution of task body (and not resolved there)
  - task is completed
  - exception is <u>not</u> propagated
- during task rendezvous
  - this is the really tricky part

Exceptions During Task Communication

- If the **called** task terminates abnormally

  exception TASKING_ERROR is raised in **calling** task at the point of the entry call

- If an entry call is made for entry of a task that becomes completed before accepting the entry

  exception TASKING_ERROR is raised in **calling** task at the point of the entry call

- If the **calling** task terminates abnormally

  no exception propagates to the **called** task

- If an exception is raised in **called** task within an **accept** (and not handled there locally)

  the same exception is raised in the **calling** task at the point of the entry call
  (even if exception is later handled outside of the accept in the called task)

Tasking Example

```
procedure  Critical_Code  is

        Failure : exception;
        -------------
        task  Monitor  is
                entry Do_Something;
        end Monitor;
        task  body  Monitor  is
        ...
        begin
                accept  Do_Something   do
                        ...
                        raise  Failure;
                        ...
                end   Do_Something;
                ...
        exception   -- exception handled here
                when Failure =>
                        Termination_Message;
        end Monitor;
        -------------
begin  -- Critical_Code
        ...
        Monitor.Do_Something;
        ...
exception   -- same exception will be handled here
        when Failure =>
                Critical_Problem_Message;

end   Critical_Code;
```

Outline

- Overview
- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions
- Turning off exception checking
- Tasking exceptions

=> **More examples**

- Summary

More Examples

- Interactive data input
- Propagating exception out of scope and back in
- Keeping a task alive

Interactive Data Input

```
with TEXT_IO; use TEXT_IO;
procedure Get_Input (Number : out integer) is

        subtype Input_Type is integer range  0..100;
        package Int_io is new INTEGER_IO (Input_Type);
        In_Number : Input_Type;

begin  -- Get_Input

        loop             -- to try again after incorrect input

                begin  -- inner block to hold exception handler

                        put ("Enter a number 0 to 100");
                        Int_io.GET (In_Number);
                        Number := In_Number;
                        exit;  -- to exit loop after correct input

                exception
                        when DATA_ERROR =>
                                put ("Try again, fat fingers!");
                                Skip_Line;    -- must clear buffer

                end;  -- inner block

        end  loop;

end  Get_Input;
```

Propagating Exception Out of Scope and Back In

```
declare
        package Container is
                procedure Has_Handler;
                procedure Raises_Exception;
        end Container;
          ----------
        procedure Not_in_Package is
        begin
                Container.Raises_Exception;
        exception
                when others => raise;
        end  Not_in_Package;
          ----------
        package  body  Container  is
                Crazy : exception;
                procedure  Has_Handler  is
                begin
                        Not_in_Package;
                exception
                        when Crazy => Tell_Everyone;
                end  Has_Handler;
                procedure   Raises_Exception  is
                begin
                        raise Crazy;
                end   Raises_Exception;
        end Container;
begin
        Container.Has_Handler;
end;
```

Keeping a Task Alive

```
task Monitor is
      entry Do_Something;
end Monitor;

task body Monitor is
begin
      loop        -- for never-ending repetition
            ...
            select
                  accept Do_Something do

                        begin  -- block for exception handler
                              ...
                              raise Failure;
                              ...
                        exception
                              when Failure => Recover;
                        end;  -- block

                  end Do_Something;  -- exception must be
                                     -- lowered before exiting
                  ...
            end select;
            ...
      end  loop;

exception
      when others =>
            Termination_Message;
end Monitor;
```

Outline

- Overview
- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions
- Turning off exception checking
- Tasking exceptions
- More examples

=> **Summary**

Summary

- Exception handling principles are consistent
- Suppression of exception checking will usually do more harm than good
- Use of exceptions must become a habit to be useful

David A. Cook
INSTRUCTOR COMPUTER SCIENCE
U S AIR FORCE ACADEMY CO 80840

472-3590
AV 259-3590
DFCS

HOME 472 693
QTRS 4206F
USAFA CO 8084

Ada* Tasking
Abstraction of Process

Captain David A. Cook
U.S. Air Force Academy

*Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

ADA TASKING

- OVERVIEW

  DEFINE ADA TASKING

  DEFINE SYNCHRONIZATION MECHANISM

  EXAMPLES

ADA TASKING

TASK DEFINITION

- A PROGRAM UNIT FOR CONCURRENT EXECUTION
- NEVER A LIBRARY UNIT
- MASTER IS A ...
  - LIBRARY PACKAGE
  - SUBPROGRAM
  - BLOCK STATEMENT
  - OTHER TASK

ADA TASKING

SYNCHRONIZATION MECHANISMS

- GLOBAL VARIABLES
- RENDEZVOUS

MAIN PROGRAM I~~N~~S A TASK

CALLER REQUESTS SERVICE

1. IMMEDIATE REQUEST
2. WAIT FOR A WHILE
3. WAIT FOREVER

CALLEE PROVIDES SERVICE

1. IMMEDIATE RESPONSE
2. WAIT FOR A WHILE
3. WAIT FOREVER

SERVICE IS REQUESTED WITH AN ENTRY CALL STATEMENT

SERVICE IS PROVIDED WITH AN ACCEPT STATEMENT

ADA TASKING

SELECT STATEMENTS PROVIDE ABILITY TO PROGRAM THE DIFFERENT REQUEST AND PROVIDE MODES

GUARDS ARE "IF STATEMENTS" FOR PROVIDING SERVICE [True or False Condition]

TERMINATION IS AN ALTERNATIVE IF A SERVICE IS NO LONGER NEEDED

TASK MASTERS

EACH TASK MUST DEPEND ON A MASTER

A MASTER CAN BE A TASK, A CURRENTLY EXECUTING BLOCK STATEMENT, A CURRENTLY EXECUTING SUBPROGRAM, OR A LIBRARY PACKAGE.

PACKAGES DECLARED INSIDE ANOTHER PROGRAM UNIT CANNOT BE MASTERS.

THE MASTER OF A TASK IS DETERMINED BY THE CREATION OF THE TASK OBJECT.

A BLOCK, TASK, OR SUBPROGRAM CANNOT BE LEFT UNTIL ALL OF ITS DEPENDENTS ARE TERMINATED.

FOR THE MAIN PROGRAM, TERMINATION DOES NOT DEPEND ON TASK WHOSE MASTER IS A LIBRARY PACKAGE.

ACTUALLY, THE 1815A DOES NOT DEFINE IF TASKS THAT DEPEND ON LIBRARY PACKAGES ARE REQUIRED TO TERMINATE!!

WHEN DOES A TASK START?

TASKS ARE ACTIVATED AFTER THE ELABORATION OF THE DECLARATIVE PART.

EFFECTIVELY, ACTIVATION IS AFTER THE DECLARATIVE PART, AND IMMEDIATELY AFTER THE 'BEGIN' STATEMENT, BUT BEFORE ANY OTHER STATEMENT.

THE PURPOSE OF THIS IS TO ALLOW THE EXCEPTION HANDLER TO SERVICE TASK EXCEPTION.

```
Task type T1 is ....
Obj : T1;

begin
    declare
    New_Obj:T1;

    begin
       null;
    end;

...
...
end;
```

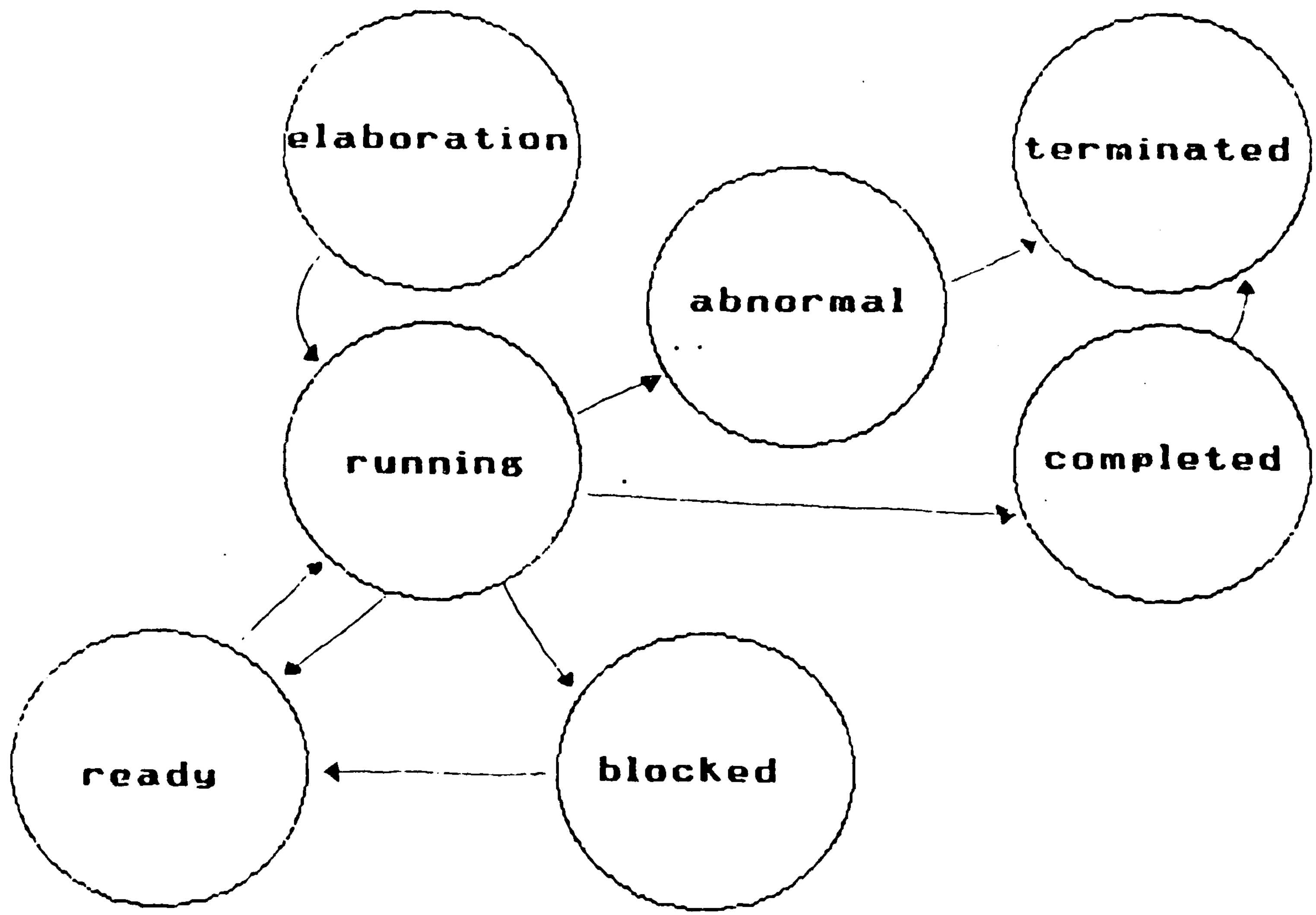
TASKS OBJECTS ACCESSED BY ALLOCATORS DO THINGS A LITTLE BIT DIFFERENTLY

NORMALLY, THE SCOPE OF A TASK OBJECT DETERMINES ITS MASTER

FOR AN ACCESS TYPE, THE MASTER IS DETERMINED BY THE ACCESS TYPE DEFINITION

ACTIVATION FOR ACCESSED TASKS OCCURS IMMEDIATELY UPON THE ASSIGNMENT OF A VALUE TO THE ACCESS OBJECT

```
Task Type T1 is...
Obj : T1;
Type T1_Ptr is access T1;
Ptr_Obj : T1_Ptr := new T1;

begin
    declare
    New_Ptr_Obj:T1_Ptr:=new T1;

    begin
       null;
    end;
...
...
end;
```


elaboration
running
abnormal
terminated
completed
ready
blocked

ELABORATION	- DECLARATIVE PART
RUNNING	- TASK HAS PROCESSOR
READY	- TASK IS AVAILABLE FOR PROCESSOR, AND HAS ALL RESOURCES TO RUN
BLOCKED	- TASK IS EITHER WAITING FOR A CALL, OR WAITING FOR CALL TO BE ANSWERED
COMPLETED	- AT END, OR EXCEPTION
TERMINATED	- COMPLETED, AND DEPENDENT TASKS ALSO TERMINATED
ABNORMAL	- TASK WAS ABORTED

```
task [type] [is
   {entry_declaration}
   {representation_clause}
end [task_simple_name] ]


task body task_simple_name is
   [declarative_part]
begin
   [sequence_of_statements]
[exception
   exception_handler
  {exception_handler}]
end [task_simple_name];
```

ACCEPT STATEMENT

THE ACCEPT STATEMENT ALLOWS AN UNKNOWN CALLER TO CALL AN ENTRY.

THERE CAN BE IN AND/OR OUT PARAMETERS

THE CONSTRUCT IS 'ACCEPT.....DO'

DURING THE ACCEPT, THE CALLING UNIT IS SUSPENDED. THUS, A LONG ACCEPT SLOWS DOWN THE SYSTEM.

A GOOD APPROACH IS TO USE THE ACCEPT SIMPLY TO COPY IN OR OUT DATA, AND ALLOW THE CALLER TO CONTINUE.

SIMPLEST FORM OF TASK ENTRY

ACCEPT

```
TASK T1 IS
     ENTRY ENTRY1;
END T1;
.
.
TASK BODY T1 IS
  BEGIN
    LOOP

          ACCEPT ENTRY1 DO
               <SOS>
          END ENTRY1;
     <SOS>

     END LOOP;
  END T1;
--WAIT FOREVER FOR CALL TO ENTRY1
```

```
TASK T1 IS
  ENTRY ACTION (DATA : SOME_TYPE);
END T1;

TASK BODY T1 IS

  BEGIN
    LOOP
      ACCEPT ACTION(DATA:SOME_TYPE) DO
        --SOME LONG PROCESS USING DATA
        --  OCCURS HERE
      END ACTION;
    END LOOP;
  END T1;

--NO EXITS OR GOTOS ALLOWED IN ACCEPT,
--  BUT A RETURN IS ALLOWED
```

```
TASK T1 IS
  ENTRY ACTION (DATA : SOME_TYPE);
END T1;


TASK BODY T1 IS
  LOCAL : SOME_TYPE;
  BEGIN
    LOOP
      ACCEPT ACTION(DATA:SOME_TYPE) DO
        LOCAL := DATA;
      END ACTION;
       --PUT PROCESS ON LOCAL HERE
    END LOOP;
  END T1;
--WHEN THIS CAN BE DONE, IT WILL SPEED
--UP THE SYSTEM.
```

```
TASK T1 IS
  ENTRY ACTION(DATA:A_TYPE);
  ENTRY RESULT(DATA :OUT A_TYPE);
END T1;

TASK BODY T1 IS
  LOCAL : A_TYPE;
  BEGIN
    LOOP
      ACCEPT ACTION(DATA:A_TYPE) DO
        LOCAL := DATA;
      END ACTION;
      --PROCESS ON LOCAL HERE
      ACCEPT R[illegible]T(DATA:OUT A_TYPE) DO
        DATA :
      END RES
    END LOOP;
  END T1;
```

```
TASK T1 IS
     ENTRY ENTRY1;
END T1;
.
.
TASK BODY T1 IS
  BEGIN
    LOOP

           ACCEPT ENTRY1; --'SYNC' CALL ONLY
     <SOS>

     END LOOP;
  END T1;
--WAIT FOREVER FOR CALL TO ENTRY1


--EVEN IF ENTRY1 HAS PARAMETERS ASSOCIATED WITH
--    IT, THE ACCEPT BLOCK DOES NOT HAVE TO HAVE A
--    SEQUENCE OF STATEMENTS
```

SELECT STATEMENT

USED BY THE TASK TO ALLOW OPTIONS

SIMPLEST FORM IS THE SELECTIVE WAIT (WAIT FOREVER)

```
TASK T1 IS
    ENTRY ENTRY1;
    ENTRY ENTRY2;
END T1;
.
.
.
TASK BODY T1 IS
 BEGIN
   LOOP
      SELECT
         ACCEPT ENTRY1 DO
           <SOS>
         END ENTRY1;
         <SOS>
      OR
         ACCEPT ENTRY2 DO
           <SOS>
         END ENTRY2;
         <SOS>

     --AS MANY 'OR' AND ACCEPT CLAUSES AS NEEDED

      END SELECT;
    END LOOP;
 END T1;
--WAIT FOR EITHER ENTRY1 OR ENTRY2
```

SELECTIVE WAIT WITH ELSE (DON'T WAIT AT ALL)

```
TASK T1 IS
    ENTRY ENTRY1;
END T1;
.
.
.
TASK BODY T1 IS
  BEGIN
   LOOP
    SELECT
      ACCEPT ENTRY1 DO
         <SOS>
      END ENTRY1;
      <SOS>
    ELSE
      <SOS>
    END SELECT;
   END LOOP;
  END T1;
```

IF THERE IS NOT A CALLER WAITING <u>RIGHT NOW</u>,
DO THE ELSE PART.

SELECTIVE WAIT WITH ELSE, MULTIPLE
ACCEPTS

```
TASK T1 IS
     ENTRY ENTRY1;
     ENTRY ENTRY2;
END T1;


TASK BODY T1 IS
  BEGIN
    LOOP
      SELECT
         ACCEPT ENTRY1 DO
           <SOS>
         END ENTRY1;
         <SOS>
      OR
         ACCEPT ENTRY2 DO
           ...
      -- AS MANY 'OR' AND 'ACCEPT' CLAUSES AS NEEDED

      ELSE
         <SOS>;
      END SELECT;
    END LOOP;
  END T1;
```

SELECT WITH DELAY ALTERNATIVE
(WAIT A FINITE TIME)

```
TASK BODY T1 IS
BEGIN
  LOOP
    SELECT
      ACCEPT ENTRY1 DO....
    [OR
      ACCEPT ENTRY2.....]
    OR
      DELAY 15.0; --SECONDS
      <SOS>;
    END SELECT;
  END LOOP;
END T1;
```

IF ENTRY1 CALLED WITHIN 15 SECONDS, THEN YOU ACCEPT THE CALL. OTHERWISE, AFTER 15 SECONDS YOU WILL DO SOMETHING.

'DELAY' RULES

YOU MAY HAVE SEVERAL ALTERNATIVES WITH A DELAY STATEMENT.

SINCE DELAYS CAN BE STATIC, THE SHORTEST DELAY ALTERNATIVE WILL BE SELECTED.

ZERO AND NEGATIVE DELAYS ARE LEGAL.

YOU MAY NOT HAVE AN ELSE PART WITH A DELAY, SINCE THE DELAY WOULD NEVER BE ACCEPTED.

'DELAY' RULES

YOU MAY HAVE SEVERAL ALTERNATIVES WITH A DELAY STATEMENT.

SINCE DELAYS CAN BE STATIC, THE SHORTEST DELAY ALTERNATIVE WILL BE SELECTED.

ZERO AND NEGATIVE DELAYS ARE LEGAL.

YOU MAY NOT HAVE AN ELSE PART WITH A DELAY, SINCE THE DELAY WOULD NEVER BE ACCEPTED.

SELECT WITH DELAY ALTERNATIVE
(WAIT A FINITE TIME)

```
TASK BODY T1 IS
BEGIN
  LOOP
    SELECT
      ACCEPT ENTRY1 DO....
    [OR
      ACCEPT ENTRY2.....]
    OR
      DELAY <EXPRESSION>;
      <SOS>;
    OR
      DELAY <EXPRESSION>;
      <SOS>;

    --SHORTEST DELAY WILL GET CHOSEN

    END SELECT;
  END LOOP;
END T1;
```

GUARDS CAN BE USED ON ANY ACCEPT STATEMENT

```
...
...
...
  WHEN SOME_CONDITION =>
     ACCEPT ENTRY1 ......
```

IF THERE IS NO GUARD, THE ACCEPT STATEMENT IS SAID TO BE OPEN.

IF THERE IS A GUARD, AND THE WHEN CONDITION IS TRUE, THE ACCEPT IS ALSO OPEN.

FALSE GUARD STATEMENTS ARE SAID TO BE CLOSED.

OPEN ALTERNATIVES ARE CONSIDERED. IF THERE IS MORE THAN ONE, THEN ONE IS SELECTED ARBITRARILY.

IF THERE ARE NO OPEN ALTERNATIVES (AND NO ELSE PART), THE EXCEPTION PROGRAM_ERROR IS RAISED.

TERMINATION

WHEN A TASK HAS COMPLETED ITS SEQUENCE OF STATEMENTS, ITS STATUS IS COMPLETED

ADDITIONALLY, THERE IS AN OPTION THAT ALLOWS A TASK TO TERMINATE.

```
SELECT
  ACCEPT ENTRY1 DO .....
[OR
  ACCEPT ENTRY2 DO.....]
 OR
  TERMINATE;
END SELECT;
```

THIS MAY NOT BE USED WITH EITHER THE THE DELAY OR AN ELSE CLAUSE.

SINCE THIS IS USED ONLY WITH A 'WAIT FOREVER' TASK, THIS OPTION ALLOWS A TASK THAT IS WAITING FOREVER TO TERMINATE IF ITS PARENT IS ALSO READY TO QUIT.

REMEMBER....

Tasks are Non-deterministic

```
select
    accept ENTRY1;
or
    accept ENTRY2;
```

Might always take ENTRY1 !!!!

KILLING A TASK

OFTEN, A 'TERMINATE' ALTERNATIVE IS NOT SUFFICIENT.

A PARENT MAY KILL DEPENDENT TASKS (OR ITSELF) USING THE ABORT STATEMENT.

THIS SHOULD ONLY BE USED IN VERY RARE CIRCUMSTANCES.

A BETTER METHOD IS TO USE AN ENTRY TO 'ACCEPT' A SHUTDOWN CALL.

IF YOU HAVE ACCEPTED A 'SHUTDOWN' CALL, THEN IT IS OK TO ABORT YOURSELF.

```
TASK BODY T1 IS
BEGIN
 LOOP        -- THE ENDLESS LOOP OF THE
             --  TASK STARTS HERE
             -- EXIT LOOP TO TERMINATE
   SELECT
             -- THE REQUIRED ACCEPT
             --  STATEMENTS ARE CODED HERE
   OR
             ACCEPT SHUTDOWN;
             --SPECIAL FINAL ACTIONS HERE
             EXIT; -- EXITS LOOP, ENDS TASK
   OR
             TERMINATE; -- FOR CASES WHERE
                   -- SHUTDOWN NOT CALLED
   END SELECT;
 END LOOP;
END T1;
```

PROBLEMS WITH PARALLELISM

MULTIPLE 'THREADS OF CONTROL' CAN CAUSE PROBLEMS IF TWO PROCESSES ARE TRYING TO ACCESS AND UPDATE ONE PIECE OF INFORMATION AT THE SAME TIME.

PRAGMA SHARED

```
    MY-OBJECT : SOME-TYPE;
    PRAGMA SHARED (MY-OBJECT);
```

ENFORCES MUTUALLY EXCLUSIVE ACCESS

ONLY WORKS FOR SCALAR AND ACCESS TYPES

SEMAPHORES CAN ALSO BE USED TO CONTROL ACCESS TO AN OBJECT

-PROMOTES 'POLLING'

ENCAPSULATING A DATA ITEM WITHIN A TASK IS A BETTER METHOD

```
TASK SEMAPHORE IS
    ENTRY P; --GET RESOURCE
    ENTRY V; --RELEASE
END SEMAPHORE;

TASK BODY SEMAPHORE IS
     AVAILABLE : BOOLEAN := TRUE;
BEGIN
    LOOP
        SELECT
            WHEN AVAILABLE
            ACCEPT P DO
                 AVAILABLE := FALSE;
            END P;
        OR
            WHEN NOT AVAILABLE
            ACCEPT V DO
                 AVAILABLE := TRUE;
            END V;
        OR
            TERMINATE;
    END LOOP;
END SEMAPHORE;
```

```
TASK SPECIAL_OPS IS
   ENTRY ASSIGN ( OBJECT : IN SOME_TYPE );
   ENTRY RETRIEVE ( OBJECT : OUT SOME_TYPE);
END SPECIAL_OPS;

 TASK BODY SPECIAL_OPS IS
     THE_OBJECT : SOME_TYPE;
     BEGIN
        LOOP
          SELECT
           ACCEPT ASSIGN(OBJECT:IN SOME_TYPE)DO
              THE_OBJECT := OBJECT;
            END ASSIGN;
         OR
           ACCEPT RETRIEVE(OBJECT:OUT SOME_TYPE)DO
              OBJECT := THE_OBJECT;
           END RETRIEVE;
         OR
           TERMINATE;
          END SELECT;
        END LOOP;
END SPECIAL_OPS;
```

CALLING A TASK ENTRY

WHEN YOU CALL A TASK, YOU MUST KNOW THE TASK NAME.

THERE ARE THREE TYPES

- ENTRY CALLS (WAIT FOREVER)
- TIMED ENTRY CALLS (WAIT FOR SPECIFIED TIME)
- CONDITIONAL ENTRY CALLS (DON'T WAIT AT ALL)

CALL AND WAIT FOREVER

TO CALL AN ENTRY, SPECIFY THE TASK NAME AND THEN THE ENTRY NAME

```
BEGIN
...
  T1.ENTRY1(DATA);
```

TIMED ENTRY CALL
(WAIT FOR A FINITE TIME)

```
SELECT
   T1.ENTRY1(DATA);
      <SOS>
OR
   DELAY 60;
   <SOS>
END SELECT;
```

YOU CANNOT USE AN 'OR' TO CALL TWO (OR MORE) TASK ENTRIES!!!

THIS WOULD BE EQUIVALENT TO STANDING IN TWO DIFFERENT LINES AT ONCE.

CONDITIONAL ENTRY CALLS
(DON'T WAIT AT ALL)

```
SELECT
   T1.ENTRY1(DATA);
   <SOS>
ELSE
   <SOS>
END SELECT;
```

NOTICE THE 'ORTHOGONALITY' OR THE SELECT STATEMENT. IT IS USED IN EITHER A TASK ENTRY CALL OR AN ACCEPT STATEMENT.

ALSO NOTICE THAT INSTEAD OF
  'ACCEPT...BEGIN...END ACCEPT;
IT IS
  'ACCEPT...DO....END ENTRY_NAME;

WHY???

TASK ATTRIBUTES

T'CALLABLE - RETURNS BOOLEAN VALUE
TRUE -TASK CALLABLE,
FALSE -TASK COMPLETED,
ABNORMAL OR TERMINATED

T'TERMINATED - BOOLEAN VALUE
TRUE IF TERMINATED

E'COUNT - RETURNS AN UNIVERSAL
INTEGER INDICATING THE
NUMBER OF ENTRY CALLS
QUEUED FOR ENTRY E.

-AVAILABLE ONLY WITHIN
TASK T ENCLOSING E

TASK PRIORITIES

PRAGMA PRIORITY (STATIC_EXPRESSION);

USED TO REPRESENT DEGREE OF RELATIVE URGENCY.

IF TWO TASKS ARE READY, THEN THE TASK WITH THE HIGHER PRIORITY RUNS.

ALTHOUGH PRIORITIES ARE STATIC, TASK RENDEZVOUS ARE DYNAMIC. WHEN TASKS ARE IN RENDEZVOUS, THE PRIORITY IS THE HIGHER OF THE CALLER AND THE CALLEE.

SYNCHRONIZATION OF DATA

```
TASK SYNC IS
   ENTRY UPDATE ( DATA : IN DATA_TYPE);
   ENTRY READ   ( DATA :OUT DATA_TYPE);
END SYNC;

TASK BODY SYNC IS
  LOCAL : DATA_TYPE;
  BEGIN
    LOOP

      SELECT
         ACCEPT UPDATE(DATA : IN DATA_TYPE) DO
            LOCAL := DATA;
         END UPDATE;
      OR
         TERMINATE;
      END SELECT;

      SELECT
         ACCEPT READ (DATA : OUT DATA_TYPE) DO
            DATA := LOCAL;
         END READ;
      OR
         TERMINATE;
      END SELECT;

    END LOOP;
  END SYNC;
```

FAMILIES OF ENTRIES

```
TYPE URGENCY IS (LOW, MEDIUM, HIGH);

TASK MESSAGE IS
     ENTRY RECEIVE(URGENCY) (DATA : DATA_TYPE);
END MESSAGE;

TASK BODY MESSAGE IS
  BEGIN
    LOOP
      SELECT
        ACCEPT RECEIVE(HIGH) (DATA:DATA_TYPE) DO
          ...
        END RECEIVE;
      OR
        WHEN RECEIVE(HIGH)'COUNT = 0 =>
        ACCEPT RECEIVE(MEDIUM) (DATA:DATA_TYPE) DO
          ...
        END RECEIVE;
      OR
        WHEN RECEIVE(HIGH)'COUNT+RECEIVE(MEDIUM)'COUNT=0 =>
        ACCEPT RECEIVE(LOW) (DATA:DATA_TYPE) DO
          ...
        END RECEIVE;
      OR
        DELAY 1.0; -- SHORT WAIT
   END MESSAGE;
```

SAME THING, WITH NO GUARDS

```
TYPE URGENCY IS (LOW, MEDIUM, HIGH);

TASK MESSAGE IS
     ENTRY RECEIVE(URGENCY) (DATA : DATA_TYPE);
END MESSAGE;

TASK BODY MESSAGE IS
  BEGIN
    LOOP
      SELECT
           ACCEPT RECEIVE(HIGH) (DATA:DATA_TYPE) DO
           ...
           END RECEIVE;
      ELSE
           SELECT
              ACCEPT RECEIVE(MEDIUM) (DATA:DATA_TYPE) DO
              ...
              END RECEIVE;
           ELSE
              SELECT
                 ACCEPT RECEIVE(LOW) (DATA:DATA_TYPE) DO
                 ...
                 END RECEIVE;
              OR
                 DELAY 1.0; -- SHORT WAIT
            END SELECT;
         END SELECT;
      END SELECT;
   END MESSAGE;
```

REPRESENTATION SPECIFICATIONS

LENGTH CLAUSE

T'STORAGE_SIZE

```
TASK TYPE T1 IS
    ENTRY ENTRY_1;
    FOR T1'STORAGE_SIZE USE
         2000*SYSTEM.STORAGE_UNIT);
END T1;
```

THE PREFIX T DENOTES A TASK TYPE.

THE SIMPLE EXPRESSION MAY BE STATIC, AND IS USED TO SPECIFY THE NUMBER OS STORAGE UNITS TO BE RESERVED OR FOR EACH ACTIVATION (NOT THE CODE) OF THE TASK.

ADDRESS CLAUSE

```
TASK TYPE T1 IS
     ENTRY ENTRY_1;
     FOR T1 USE AT 16#167A#;
END T1;
```

IN THIS CASE, THE ADDRESS SPECIFIES THE ACTUAL LOCATION IN MEMORY WHERE THE MACHINE CODE ASSOCIATED WITH T1 WILL BE PLACED.

```
TASK T1 IS
     ENTRY ENTRY_1;
     FOR ENTRY_1 USE AT 16#40#;
END T1;
```

IF THIS CASE, ENTRY_1 WILL BE MAPPED TO HARDWARE INTERRUPT 64.

ONLY IN PARAMETERS CAN BE ASSOCIATED WITH INTERRUPT ENTRIES.

AN INTERRUPT WILL ACT AS AN ENTRY CALL ISSUED BY THE HARDWARE, WITH A PRIORITY HIGHER THAN ANY USER-DEFINED TASK.

DEPENDING UPON THE IMPLEMENTATION, THERE CAN BE MANY RESTRICTIONS UPON THE TYPE OF CALL TO THE INTERRUPT, AND UPON THE TERMINATE ALTERNATIVES.

NOTE: YOU CAN DIRECTLY CALL AN INTERRUPT ENTRY.

TASKS AT DIFFERENT PRIORITIES

GIVEN 5 TASKS, 3 OF VARYING PRIORITY, 1 TO BE INTERRUPT DRIVEN, AND 1 THAT WILL BE TIED TO THE CLOCK.

```
PROCEDURE HEAVY_STUFF IS

   TASK HIGH_PRIORITY IS
        PRAGMA PRIORITY(50);  --OR AS HIGH AS SYSTEM ALLOWS
        ENTRY POINT;
   END HIGH_PRIORITY;

   TASK MEDIUM_PRIORITY IS
        PRAGMA PRIORITY(25);
        ENTRY POINT;
   END MEDIUM_PRIORITY;

   TASK LOW_PRIORITY IS
        PRAGMA PRIORITY(1);
        ENTRY POINT;
   END LOW_PRIORITY;

   TASK INTERRUPT_DRIVEN IS
        ENTRY POINT;
        FOR POINT USE AT 16#61#; --INTERRUPT 97
   END INTERRUPT_DRIVEN;

   TASK CLOCK_DRIVEN IS
        --THERE ARE TWO WAYS TO DO THIS

        --FIRST WAY IS TO HAVE ANOTHER TASK MONITOR
        -- THE CLOCK, AND CALL CLOCK_DRIVEN.CALL
        -- EVERY TIME UNIT.
        ENTRY CALL;

        --SECOND WAY IS TO ACTUALLY TIE CALL TO AN
        -- CLOCK INTERRUPT, AND LET CALL DETERMINE WHEN
        -- HE WISHES TO PERFORM AN ACTION
        FOR CALL USE AT 16#32#; --ASSUME INTERRUPT 50
                                -- IS A CLOCK INTERRUPT
     END CLOCK_DRIVEN;
END HEAVY_STUFF;
```

```
TASK QUEUE IS
    ENTRY INSERT(DATA : IN DATA_TYPE);
    ENTRY REMOVE(DATA :OUT DATA_TYPE);
END QUEUE;

TASK BODY QUEUE IS
   HEAD,TAIL : INTEGER := 0;
   Q : ARRAY (1..100) OF DATA_TYPE;
   BEGIN
     LOOP
       SELECT
         WHEN TAIL - HEAD + 1 /= 0 AND THEN
              TAIL - HEAD + 1 /= 100 =>
         ACCEPT INSERT(DATA : IN DATA_TYPE) DO
              IF HEAD = 0 THEN HEAD := 1; END IF;
              IF TAIL = 100 THEN TAIL := 0; END IF;
              TAIL := TAIL + 1;
              Q(TAIL) := DATA;
         END INSERT;
       OR
         WHEN HEAD /= 0 =>
         ACCEPT REMOVE(DATA :OUT DATA_TYPE) DO
              DATA := Q(HEAD);
              IF HEAD = TAIL THEN
                 HEAD := 0;
                 TAIL := 0;
              ELSE
                 HEAD := HEAD + 1;
                 IF HEAD > 100 THEN HEAD := 1; END IF;
              END IF;
           END REMOVE;
       OR
           TERMINATE;
       END SELECT;
     END LOOP;
   END QUEUE;
```

```
TASK TYPE QUEUE IS
    ENTRY INSERT(DATA : IN DATA_TYPE);
    ENTRY REMOVE(DATA :OUT DATA_TYPE);
END QUEUE;

TASK BODY QUEUE IS
   HEAD,TAIL : INTEGER := 0;
   Q : ARRAY (1..100) OF DATA_TYPE;
   BEGIN
     LOOP
       SELECT
         WHEN TAIL - HEAD + 1 /= 0 AND THEN
              TAIL - HEAD + 1 /= 100 =>
         ACCEPT INSERT(DATA : IN DATA_TYPE) DO
              IF HEAD = 0 THEN HEAD := 1; END IF;
              IF TAIL = 100 THEN TAIL := 0; END IF;
              TAIL := TAIL + 1;
              Q(TAIL) := DATA;
         END INSERT;
       OR
         WHEN HEAD /= 0 =>
         ACCEPT REMOVE(DATA :OUT DATA_TYPE) DO
              DATA := Q(HEAD);
              IF HEAD = TAIL THEN
                 HEAD := 0;
                 TAIL := 0;
              ELSE
                 HEAD := HEAD + 1;
                 IF HEAD > 100 THEN HEAD := 1; END IF;
              END IF;
           END REMOVE;
       OR
           TERMINATE;
       END SELECT;
     END LOOP;
   END QUEUE;

   MY_QUEUE, YOUR_QUEUE : QUEUE; -- TWO TASKS
```

```
GENERIC
DATA_TYPE : PRIVATE;
QUEUE_SIZE: POSITIVE := 100;

PACKAGE QUEUE_PACK IS

TASK QUEUE IS
    ENTRY INSERT(DATA : IN DATA_TYPE);
    ENTRY REMOVE(DATA :OUT DATA_TYPE);
END QUEUE;

PACKAGE BODY QUEUE_PACK IS
TASK BODY QUEUE IS
   HEAD,TAIL : INTEGER := 0;
   Q : ARRAY (1..QUEUE_SIZE) OF DATA_TYPE;
   BEGIN
     LOOP
       SELECT
         WHEN TAIL - HEAD + 1 /= 0 AND THEN
              TAIL - HEAD + 1 /= QUEUE_SIZE =>
         ACCEPT INSERT(DATA : IN DATA_TYPE) DO
              IF HEAD = 0 THEN HEAD := 1; END IF;
              IF TAIL = QUEUE_SIZE THEN TAIL := 0; END IF;
              TAIL := TAIL + 1;
              Q(TAIL) := DATA;
         END INSERT;
       OR
         WHEN HEAD /= 0 =>
         ACCEPT REMOVE(DATA :OUT DATA_TYPE) DO
              DATA := Q(HEAD);
              IF HEAD = TAIL THEN
                 HEAD := 0;
                 TAIL := 0;
              ELSE
                 HEAD := HEAD + 1;
                 IF HEAD> QUEUE_SIZE THEN HEAD := 1; END IF;
              END IF;
          END REMOVE;
       OR
          TERMINATE;
       END SELECT;
     END LOOP;
   END QUEUE;


PACKAGE NEW_QUEUE IS NEW QUEUE_PACK(MY_RECORD, 250);
PACKAGE OLD_QUEUE IS NEW QUEUE_PACK(INTEGER);
```

```
PROCEDURE INSERT_INTEGER (DATA : IN INTEGER ) RENAMES
    OLD_QUEUE.INSERT;

PROCEDURE REMOVE_INTEGER (DATA :OUT INTEGER ) RENAMES
    OLD_QUEUE.REMOVE;
```

```
PROCEDURE SPIN (R : RESOURCE) IS
BEGIN
  LOOP
    SELECT
       R.SEIZE;
       RETURN;
    ELSE
       NULL;  --BUSY WAITING
    END SELECT;
  END LOOP;
END;
```

--OR--

```
PROCEDURE SPIN (R : RESOURCE) IS
BEGIN
   R.SEIZE;
   RETURN;
END;
```

ADA TASKING

SCENARIO I

"THE GOLDEN ARCHES"

MCD TASKS :

- SERVICE PROVIDED : FOOD
- SERVICE REQUESTED : NONE

GONZO TASKS :

- SERVICE PROVIDED : NONE
- SERVICE REQUESTED : FOOD

```
Task McD is
    entry SERVE(TRAY_OF : out FOOD_TYPE);
end McD;

Task GONZO;

Task Body McD is
    NEW_TRAY : FOOD_TYPE;
    function COOK return FOOD_TYPE is ......
    begin
        loop
            accept SERVE(TRAY_OF : out FOOD_TYPE) do
                TRAY_OF := COOK;
            end;
        end loop;
    end McD;
```

```
Task Body GONZO is
    MY_TRAY : FOOD_TYPE;

    procedure CONSUME(MY_TRAY:in FOOD_TYPE) is ...

    begin
        loop
            McD.SERVE ( MY_TRAY);
            CONSUME(MY_TRAY);
        end loop;
    end GONZO;
```

```
Task Body McD is
    NEW_TRAY : FOOD_TYPE;

    function COOK return FOOD_TYPE is
        ...
    end COOK;

    begin
        loop
            NEW_TRAY := COOK;
            accept SERVE(TRAY_OF:out FOOD_TYPE) do
                TRAY_OF := NEW_TRAY;
            end SERVE;
        end loop;
    end GONZO;
```

```
loop
    NEW_TRAY := COOK;
    select
        accept SERVE(TRAY_OF : out FOOD_TYPE) do
            TRAY_OF := NEW_TRAY;
        end SERVE;
    else
        null;
    end select;
end loop;
```

```
loop
    NEW_TRAY := COOK;
    select
        accept SERVE(TRAY_OF : out FOOD_TYPE) do
            TRAY_OF := NEW_TRAY;
        end SERVE;
    else or
        terminate;
    end select;
end loop;
```

```
loop
    NEW_TRAY := COOK;
    select
        accept SERVE(TRAY_OF : out FOOD_TYPE) do
            TRAY_OF := NEW_TRAY;
        end SERVE;
    or
        delay 15.0 * MINUTES;
    end select;
end loop;
```

```
loop

    select
        McD.SERVE(MY_ORDER);  CONSUME(MY_ORDER);
    else
        select
            BK.SERVE(MY_ORDER); CONSUME (MY_ORDER);
        else
            exit;
        end select;
    end select;

end loop;
```

```
loop

    select
        McD.SERVE(MY_ORDER);  CONSUME(MY_ORDER);
    or
        delay 5.0 * MINUTES;
        select
            BK.SERVE(MY_ORDER); CONSUME (MY_ORDER);
        or
            delay 5.0 * MINUTES;
            exit;
        end select;
    end select;

end loop;
```

```
loop

    select
        McD.SERVE (MY_ORDER);
    or                                        --**
        BK.SERVE(MY_ORDER);
    end select;

    CONSUME(MY_ORDER);

end loop;
```

```
loop

    select
        McD.SERVE (MY_ORDER);
    or                                          --**
        BK.SERVE(MY_ORDER);
    else
        delay 10.0 * MINUTES;
        exit;
    end select;

    CONSUME(MY_ORDER);

end loop;
```

ADA TASKING

SCENARIO II

"NO FREE LUNCH"

MCD TASK
- SERVICE PROVIDED : FOOD
- SERVICE REQUESTED: MONEY

GONZO TASK
- SERVICE PROVIDED : MONEY
- SERVICE REQUESTED: FOOD

```
Task McD is
    entry SERVE(ORDER: out FOOD_TYPE;
                COST: in  MONEY_TYPE);
end McD;

Task GONZO;



--OR

Task McD is
    entry SERVE(ORDER: out FOOD_TYPE);
end McD;

Task GONZO is
    entry PAY (COST : in MONEY_TYPE;
         PAYMENT : out MONEY_TYPE);
end GONZO;
```

```
Task Body McD is
   CASH_DRAWER, AMOUNT_PAID: MONEY_TYPE;
   NEW_ORDER  : FOOD_TYPE;
   function COOK ..........
   function CALC_COST(ORDER: in FOOD_TYPE)
      return MONEY_TYPE ..........

   begin
      loop
         NEW_ORDER := COOK;
         select
            accept SERVE(ORDER:out FOOD_TYPE) do
               ORDER := NEW_ORDER;
               COST := CALC_COST(NEW_ORDER);
               GONZO.PAY(COST, AMOUNT_PAID); --**
               CASH_DRAWER :=
                  CASH_DRAWER + AMOUNT_PAID;
            end SERVE;
         or
            delay 15.0 * MINUTES;
         end select;
      end loop;
   end McD;
```

```
Task Body GONZO IS
   ACCOUNT_BALANCE : MONEY_TYPE;
   MY_ORDER : FOOD_TYPE;
   function GO_TO_WORK return MONEY_TYPE ......

   begin
      ACCOUNT_BALANCE :=
         ACCOUNT_BALANCE + GO_TO_WORK;
      loop
         McD.SERVE(MY_ORDER);
         accept PAY (COST : in MONEY_TYPE;
                     PAYMENT:out MONEY_TYPE) do
            ACCOUNT_BALANCE :=
               ACCOUNT_BALANCE - COST;
            PAYMENT := COST;
         end PAY;
      end loop;
   end GONZO;
```

ADA TASKING

SCENARIO II A

"NO WAIT FOR THE WAITERS"

MCD TASK
- SERVICE PROVIDED : FOOD
- SERVICE REQUESTED: MONEY

GONZO TASK
- SERVICE PROVIDED : MONEY
- SERVICE REQUESTED: FOOD

MANAGER TASK
- SERVICE PROVIDED : MAKE NEW WAITER
- SERVICE REQUESTED: NONE

```
Task type McD is
    entry SERVE.....
end McD;

Task GONZO is
    entry PAY.....
end GONZO;

Task MANAGER;

Type CASHIER_POINTER is access McD;

Type REGISTER_TYPE is array (1..NO_REGISTERS)
                of CASHIER_POINTER;

THE_REGISTERS : REGISTER_TYPE
                        := (others => new McD);
```

```
Task Body McD is
    ...
    ...
    ...
    begin
        loop
            NEW_ORDER := COOK;
            select
                accept SERVE.....
                ...
                end SERVE;
            or
                delay 2,0 * MINUTES;
                exit;
            end select;
        end loop;
```

```
Task Body GONZO is
    ...
    ...
    begin
        ...
        ...
        --- Now, GONZO has to search for the open
        ---  registers, and select the one with
        ---  the shortest line
        ...
        ...
        THE_REGISTERS(MY_REGISTER).SERVE...
        ...
    end GONZO;
```

```
Task Body MANAGER is
    ...
    ...
    begin
        loop
        --The Manager will look at the queue lengths of
        --  the open registers, and, when necessary,
        --  will open registers that are currently
        --  closed
        ...
        ...
            if ...........then
                THE_REGISTERS(CLOSED_REGISTER):=
                                                new McD;
            end if;
        end loop;
    end MANAGER;
```

SCENARIO III

"A SUGAR CONE, PLEASE:

BR TASK
- SERVICE PROVIDED : ICE CREAM
- SERVICE REQUESTED: AN ORDER

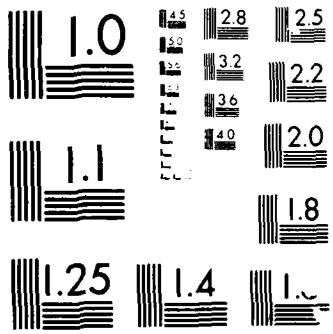
SERVOMATIC TASK
- SERVICE PROVIDED : A NUMBER

CUSTOMERS TASK
- SERVICE PROVIDED : AN ORDER
- SERVICE REQUESTED: ICE CREAM

```
Task BR is
   entry SERVE(ICE_CREAM: out DESSERT_TYPE;
end BR;

Task SERVOMATIC is
   entry TAKE(A_NUMBER: out SERVOMATIC_NUMBERS);
end SERVOMATIC;

Task type CUSTOMER_TASK is
   entry REQUEST(ORDER: out ORDER_TYPE);
enter CUSTOMER_TASK;

Type CUSTOMER is access CUSTOMER_TASK;

CUSTOMERS : array (SERVOMATIC_NUMBERS) of CUSTOMER;
```

AD-A192 074 ASEET ADVANCED ADA WORKSHOP JANUARY 1988(U) ADA JOINT PROGRAM OFFICE ARLINGTON VA JAN 88 4/5

UNCLASSIFIED F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

```
Task Body BR is
   NEXT_CUSTOMER : SERVOMATIC_NUMBERS :=
                         SERVOMATIC_NUMBERS'last;
   CUREENT_ORDER : ORDER_TYPE;
   ICE_CREAM : DESSERT_TYPE;
   function MAKE(ORDER : in ORDER_TYPE) return
                     DESSERT_TYPE is ......
   begin
     loop
        begin
          NEXT_CUSTOMER:=(NEXT_CUSTOMER+1)
                    mod SERVOMATIC_NUMBERS'last;
          CUSTOMERS(NEXT_CUSTOMER).REQUEST
                              (CURRENT_ORDER);
          ICE_CREAM := MAKE (CURRENT_ORDER);
          accept SERVE(ICE_CREAM:out DESSERT_TYPE) do
              ICE_CREAM := BR.ICE_CREAM;
          end SERVE;
        exception
          when TASKING_ERROR=>null;--customer not here
        end;
     end loop
   end;
```

```
Task Body SERVOMATIC is
   NEXT_NUMBER : SERVOMATIC_NUMBERS :=
                        SERVOMATIC_NUMBERS'first;

   begin
     loop
       accept TAKE(A_NUMBER:out SERVOMATIC_NUMBERS)d
            A_NUMBER := NEXT_NUMBER;
       end TAKE;
       NEXT_NUMBER:=(NEXT_NUMBER + 1) mod
                              SERVOMATIC_NUMBERS'last;
     end loop;

   end SERVOMATIC;
```

```
Task Body CUSTOMER_TASK is
   MY_ORDER : ORDER_TYPE := ...  -- some value
   MY_DESSERT : DESSERT_TYPE;

   begin
      accept REQUEST(ORDER:out ORDER_TYPE) do
         ORDER := MY_ORDER;
      end REQUEST;
      BR.SERVE(MY_DESSERT);
      --eat the dessert, or do whatever
   end;
```

Ada Tasking

Scenario IV

"Lets Hide the Spooler Task"

PRINTER_PACKAGE
  Action-"Hides" the print spooler
  by renaming task entry

SPOOLER Task
  Service Provided : Virtual Print
  Service Requested: Physical Print

PRINTER Task
  Service Provided : Physical Print
  Service Requested: File Name

```
Package PRINTER_PACKAGE is
    ...
    ...
    task SPOOLER is
        entry PRINT_FILE(NAME : in STRING;
                         PRIORITY : in NATURAL);
        entry PRINTER_READY;
    end SPOOLER;
    ...
    ...
    procedure PRINT (NAME : in STRING;
                     PRIORITY : in NATURAL := 10)
            renames SPOOLER.PRINT_FILE;
    end PRINTER_PACKAGE;


    Package Body PRINTER_PACKAGE is
        ...
        task PRINTER is
            entry PRINT_FILE(NAME : in STRING);
        end PRINTER;
        ...
    end PRINTER_PACKAGE;
```

```
Task Body SPOOLER is
   begin
      loop
         select
            accept PRINTER_READY do
               PRINTER.PRINT_FILE(REMOVE(QUEUE));
               --Remove would determine the next job
               -- and send it to the actual printer
            end PRINTER_READY;
         else
            null;
         end select;

         select
            accept PRINT_FILE(NAME : in STRING;
                              PRIORITY : NATURAL ) do
               INSERT (NAME, PRIORITY);
               --put name on queue or queues
               --  according to priority
            end PRINT_FILE;
         else
            null;
         end select;
      end loop;
   end SPOOLER:
```

```
Task Body PRINTER is
   begin
      loop
         SPOOLER.PRINTER_READY;
         accept PRINT_FILE (NAME : in STRING) do

            if NAME'length /= 0 then .....

               -- print the file


            else
               delay 10.0 * SECONDS;
            end if;

         end PRINT_FILE;
      end loop;

   end PRINTER;
```

```
with PRINTER_PACKAGE;

procedure MAIN is
    ...
    ...
    ...
    loop
        -- process several files

        PRINTER_PACKAGE.PRINT (A_FILE, A_PRIORITY);
        ...
        ...
    end loop;
end MAIN;
```

TASKING MINDSET

SIMPLE PROBLEM - WRITE A TASK SPEC TO LET TASK A SEND AN INTEGER TO TASK B.

SOLUTION 1 - A CALLS AN ENTRY IN B

SOLUTION 2 - B CALLS FOR AN ENTRY IN A

SOLUTION 3 - WRITE A 'BUFFER' TASK
TO CALL ENTRY IN A, GET INTEGER, AND
THEN CALL ENTRY IN B TO SEND INTEGER

SOLUTION 4 - WRITE BUFFER TASK C TO ACCEPT INTEGERS FROM A, AND ALSO ACCEPT REQUESTS FROM B

IN-CLASS EXERCISE

LET US DESIGN THE TASK SPECIFICATIONS FOR THE FOLLOWING SENARIO.

THREE TASKS HAVE ACCES TO A TYPE KNOWN AS MESSAGE_TYPE.

TASK_1 PRODUCES MESSAGES. TASK_2 CAN RECEIVE MESSAGES, HOLD THEM IN A BUFFER (IF NECESSARY), AND SENDS THEM TO TASK_3 WHEN THE DATE/TIME FIELD (PART OF MESSAGE_TYPE) SAYS TO.

```
TASK TASK_1 IS



END TASK_1;



TASK TASK_2 IS



END TASK_2;



TASK TASK_3 IS



END TASK_3;
```

TASKING EXERCISE

WRITE A MAIN PROGRAM AND TWO TASKS TO SIMULATE A HOUSE ALARM SYSTEM. THE MAIN PROGRAM IS AN INPUT SIMULATOR TO THE TASKS. ONE TASK KEEPS TRACK OF THE STATUS OF THE HOUSE. ANOTHER IS THE ACTUAL ALARM SYSTEM.

TASK 1: THE HOUSE STATUS (TASK NAME :HOUSE)
THREE ENTRIES => OK, NOT_OK, WRITE

THE ENTRIES OK AND NOT_OK SET OR RESET A FLAG THAT DETERMINES THE STATUS OF THE HOUSE. NOT_OK WILL ALSO SET A VARIABLE TO TELL YOU WHICH ALARM IS CURRENTLY GOING OFF. BOTH OK AND NOT_OK SHOULD PRINT OUT A MESSAGE VERIFYING THAT THEY WERE CALLED. THE WRITE ENTRY WILL PRINT THE STATUS OF THE HOUSE. IF THERE IS AN ALARM CURRENTLY GOING OFF, WRITE WILL TELL YOU THE ALARM NUMBER.

TASK 2: THE ALARM SYSTEM (TASK NAME: ALARM)
THREE ENTRIES => FIRE, INTRUDER, SHUTOFF

THE ALARM SYSTEM WILL ACCEPT ANY OF THE THREE ENTRY CALLS FROM THE INPUT SIMULATOR. IF THERE ARE NO ENTRY CALLS WITHIN 5 SECONDS, IT WILL CALL HOUSE.WRITE TO DISPLAY THE STATUS. FIRE AND INTRUDER EACH HAVE A PARAMETER INDICATION THE ALARM LOCATION. FIRE LOCATIONS ARE '1' THRU '9'. INTRUDER LOCATIONS ARE 'A' THRU 'Z'. FIRE AND INTRUDER SHOULD CALL HOUSE.NOT_OK (AND TELL THE HOUSE WHERE THE ALARM IS SOUNDING), AND THEN PRINT OUT A MESSAGE

MAIN PROGRAM

THE MAIN PROGRAM WILL READ IN CHARACTERS FROM THE KEYBOARD. IF THE CHARACTER IS A '1' THRU '9', CALL THE FIRE ALARM. IF THE CHARACTER IS A 'A' THRU 'Z' THEN IT CALLS THE INTRUDER ALARM. IF THE CHARACTER IS A '0'(ZERO), THE HOUSE IS RESET TO OK. IF THE CHARACTER IS A '!', THEN THE ALARM IS SHUTDOWN, AND THE PROGRAM ENDS. ALL OTHER CHARACTERS DO NOTHING.

THE HOUSE STATUS SHOULD BE OK TO START.

```
run cookie


The house is ok


The house is ok

&
Invalid character.  Try again


The house is ok

G
House alarm set to not OK at location G
Intruder in room G

The house is not ok ..alarm is off at location G


The house is not ok ..alarm is off at location G

4
House alarm set to not OK at location 4
Fire Alarm # 4 has been set off.

The house is not ok ..alarm is off at location 4

0
House alarm reset to OK.

The house is ok


The house is ok

!
The alarm has been turned off

*)
```

```
WITH TEXT_IO;
USE TEXT_IO;

PROCEDURE COOKIE  IS

CHAR : CHARACTER;

    TASK HOUSE  IS
        ENTRY OK;
        ENTRY NOT_OK (WHERE:CHARACTER);
        ENTRY WRITE;
    END HOUSE ;

    TASK ALARM  IS
        ENTRY FIRE      (LOCATION:CHARACTER);
        ENTRY INTRUDER (LOCATION:CHARACTER);
        ENTRY SHUTOFF;
    END ALARM ;
```

```
TASK BODY HOUSE  IS
    TYPE CONDITION IS (OK, NOT_OK);
    ALARM_STATUS : CONDITION := OK;
    ALARM_LOCATION : CHARACTER;

BEGIN
    LOOP
        SELECT
            ACCEPT OK DO
                ALARM_STATUS := OK;
                PUT_LINE("HOUSE ALARM RESET TO OK.");
            END OK;
        OR
            ACCEPT NOT_OK (WHERE:CHARACTER) DO
                ALARM_STATUS := NOT_OK;
                ALARM_LOCATION := WHERE;
                PUT_LINE("HOUSE ALARM SET TO NOT OK AT"&
                      "LOCATION " & ALARM_LOCATION);
            END NOT_OK;
        OR
            ACCEPT WRITE DO
                NEW_LINE;
                CASE ALARM_STATUS IS
                  WHEN OK =>PUT_LINE("THE HOUSE IS OK");
                  WHEN NOT_OK =>  PUT_LINE
                         ("THE HOUSE IS NOT OK"&
                          " ..ALARM IS OFF AT LOCATION " &
                            ALARM_LOCATION);
                END CASE;
                NEW_LINE;
            END WRITE;
        OR
            TERMINATE;
        END SELECT;
    END LOOP;
END HOUSE ;
```

```
TASK BODY ALARM  IS
BEGIN
    LOOP
        SELECT
            ACCEPT FIRE (LOCATION:CHARACTER) DO
                HOUSE.NOT_OK(LOCATION);
                PUT ("FIRE ALARM # ");
                PUT (LOCATION);
                PUT_LINE (" HAS BEEN SET OFF.");
            END FIRE;
        OR
            ACCEPT INTRUDER (LOCATION:CHARACTER) DO
                HOUSE.NOT_OK(LOCATION);
                PUT ("INTRUDER IN ROOM ");
                PUT (LOCATION);
                NEW_LINE;
            END INTRUDER;
        OR
            ACCEPT SHUTOFF;
            PUT_LINE ("THE ALARM HAS BEEN TURNED OFF");
            EXIT;
        OR
            DELAY 5.0;
            HOUSE.WRITE;
        END SELECT;
    END LOOP;
END ALARM;
```

```
BEGIN        --MAIN
    LOOP
        GET (CHAR);
        SKIP_LINE;
        CASE CHAR IS
            WHEN '1' .. '9' =>  ALARM.FIRE (CHAR);
            WHEN 'A' .. 'Z' =>  ALARM.INTRUDER (CHAR);
            WHEN 'A' .. 'Z' =>  ALARM.INTRUDER (CHAR);
            WHEN '0'        =>  HOUSE.OK;
            WHEN 'I'        =>  ALARM.SHUTOFF;
            WHEN OTHERS     =>  PUT_LINE
                           ("INVALID CHARACTER.  TRY AGAIN");
        END CASE;
        EXIT WHEN CHAR = 'I';
    END LOOP;

END COOKIE;
```

Towers of Hanoi

An example of recursion

recursion: n., see recursion.

Problem: Move disks from one tower to another tower.

Constraints:

Move only 1 disk at a time.
Place no disk on a smaller disk.

Top down design approach:

Assume a procedure to move N disks:

```
type Towers is (Middle, Left, Right);
procedure Move (N : in positive;
                From,
                To,
                Other : in Towers);
```

Use the procedure and solve the problem:

```
Move (N=>3, From  => Middle,
            To    => Left,
            Other => Right);
```

Using this approach, we can now create a complete Ada program:

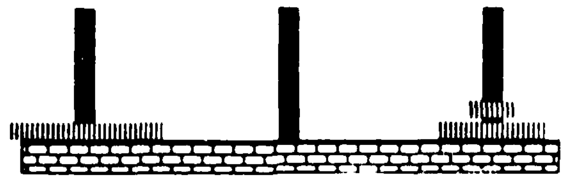
```
procedure Towers_of_Hanoi is
   type Towers is (Middle, Left, Right);
   procedure Move (N : in positive;
                   From,
                   To,
                   Other : in Towers)
      is separate;
   begin
      Move(3, From => Middle,
              To    => Left,
              Other => Right);
   end Towers_of_Hanoi;
```

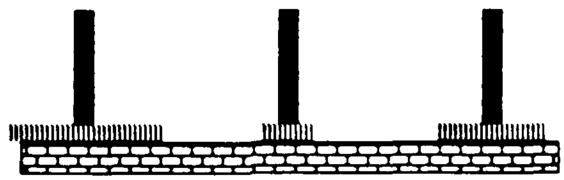
Implement the procedure in pseudocode:

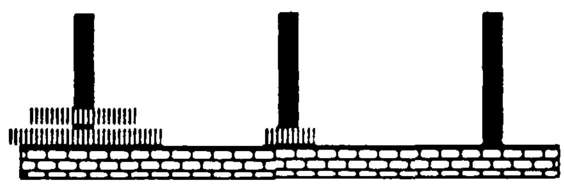
```
separate (Towers_of_Hanoi)
procedure Move (N : in positive;
                    From,
                    To,
                    Other : in Towers)
is
begin
   null;
   -- if more than one disk to move,
   -- Move(N-1, from => ________,
   --             to    => ________,
   --             other => ________);

   -- move the only disk left on 'from' to 'to'

   -- if more than one disk to move,
   -- Move(N-1, from => ________,
   --             to    => ________,
   --             other => ________);
 end Move;
```

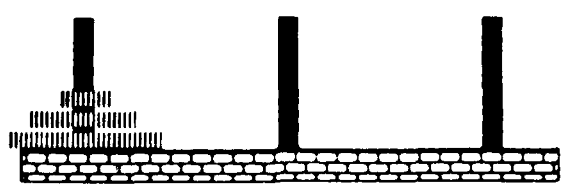
Now rewrite the procedure in Ada:

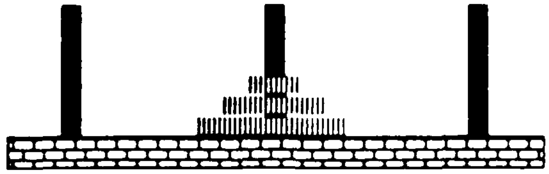
```
with Text_IO;
separate (Towers_of_Hanoi)
procedure Move (N : in positive;
                From,
                To,
                Other : in Towers)
is
begin
   if N > 1 then
     Move(N-1, From => From,
               To    => Other,
               Other => To);
    end if;

   Text_IO.put_line("Move disk from "
                  & Towers'Image(From)
                  & " tower to "
                  & Towers'Image(To)
                  & " tower.");

   if N > 1 then
     Move(N-1, From => Other,
               To    => To,
               Other => From);
   end if;

  end Move;
```

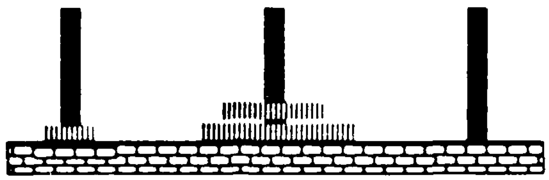


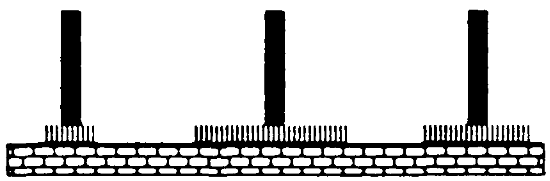


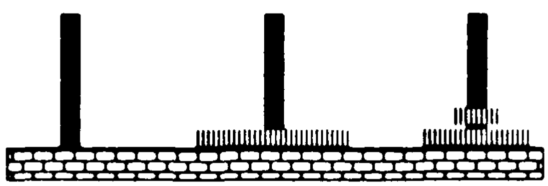












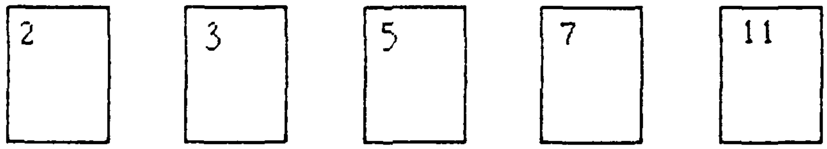
Sieve of Eratosthanes

Eratosthanes, of Alexandria, was a Greek mathematician. He developed an elegant algorithm for generating prime numbers

1. 2 is the first prime number,
2. for each positive number, N, greater than 2, if it is not divisible by any prime less then N, it is prime.

This algorithm has a natural implementation in Ada.

Imagine that a separate process is available for each prime number, that can check the "relative" primeness of a number.



We can now "pipeline" these processes with all the positive numbers, any number that makes it through the "pipe" is prime!

Create a task which feeds numbers into the pipe:

```
task Feeder;
```

Create a task template which accepts a value and checks it for primeness:

```
task type Checker is
   entry Check_It (In_Value : Positive);
end Checker;
```

But, this checker task needs to know what prime number it uses. Often we find the case in Ada tasks where the task must be initialized with information:

```
task type Checker is
   entry Who_Am_I (In_Value : Positive);
   entry Check_It    (In_Value : Positive);
end Checker;
```

Finally, we need to create new tasks when we find that a number is prime:

```
procedure Make_New_Checker
              (A_Prime_Number : in  Positive;
               New_Checker    : out Checker_Ptr);
```

We can create an operation to construct a task only by using a pointer to the new task.

There are many ways to link the checker tasks together into the "pipe". This linking determines and is determined by the manner used to pass the numbers being checked from task to task.

I chose to have each task contain the name of the next task in the pipe.

```
procedure Primes is

  task Feeder;

  type Checker;
  type Checker_ptr is access Checker;
  task type Checker is
    entry Who_Am_I (In_Value : Positive);
    entry Check_It    (In_Value : Positive);
  end Checker;

  procedure Make_New_Checker (A_Prime_Number : in  Positive;
                              New_Checker    : out Checker_Ptr);

  Front : Checker_ptr;    -- This is the front of the "pipe".

  task body Feeder is separate;

  task body Checker is separate;

  procedure Make_New_Checker (A_Prime_Number : in  Positive;
                              New_Checker    : out Checker_Ptr)
    is separate;

 begin
   null;
 end Primes;
```

```
with Text_IO, Integer_IO;
separate (Primes)
procedure Make_New_Checker (A_Prime_Number : in  Positive;
                            New_Checker    : out Checker_Ptr) is
 Result : Checker_Ptr;

begin
   -- We have been given a prime number, display it:
   Integer_IO.Put (A_Prime_Number);

   -- Make a new prime # task for it:
   Result := new Checker;
   Result.Who_Am_I (A_Prime_Number);    -- Tell the task it's prime #.

   -- Allow the task to be used in the "pipe".
   New_Checker := Result;

exception
   when Storage_Error =>
      Text_IO.Put_Line ("  Not enough room to make new tasks.");

end Make_New_Checker;
```

```
with Text_IO, Integer_IO;
separate (Primes)
task body Feeder is
   Upper_Limit : Positive;

begin

   Text_IO.Put ("Upper limit for primes? ");
   Integer_IO.Get (Upper_Limit);

   -- Generate the first prime #:
   Make_New_Checker (2, Front);

   -- Feed the "pipe":
   for Counter in 3 .. Upper_Limit loop
     Front.Check_It (Counter);
   end loop;

end Feeder;
```

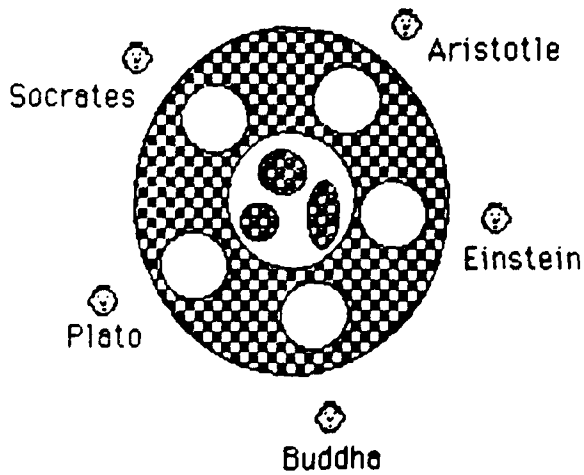
```
separate (Primes)
task body Checker is
   My_Value,
   Value_to_Check : Positive;
   Next_Checker   : Checker_Ptr;
   Prime          : Boolean;
begin

   accept Who_Am_I (In_Value : Positive) do
     My_Value := In_Value;
   end Who_Am_I;

   loop
     select
        accept Check_It (In_Value : Positive) do
           Value_to_Check := In_Value;
        end Check_It;
     or
        terminate;
     end select;

     Prime := (Value_to_Check / My_Value) * My_Value /= Value_to_Check;

     if Prime then
       if Next_Checker /= null then

         -- It's not divisible by my number, pass the value on.
         Next_Checker.Check_It (Value_to_Check);

       else

         -- It really is prime.
         Make_New_Checker (Value_to_Check, Next_Checker);

       end if;

     end if;
   end loop;

end Checker;
```

The Dining Philosophers[1]

The scenario: Five philosophers sit at a table. A lazy Susan containing dishes of Chinese food is in the center of the table. Each philosopher has a plate, but there are only five single chopsticks, one between each philosopher.

The problem: Develop a program that allows each philosopher to alternately eat and think forever. Of course, no philosopher should preclude any other from eating for an indefinite amount of time.

The constraints: Each philosopher must have control of two chopsticks to eat. But, he can only use those that were originally on either side of his plate.



[1] This problem was first stated by Edsgar Dijkstra as a challenge to the multitasking community

Approach:

Each philosopher can wait in a queue for his chopsticks. If at least one philosopher is eating, we will not have deadlock. If philosophers are not blocked from entering a queue, then we will not have indefinite postponement.

Model the chopsticks as counting semaphores.

Do not make all the philosophers right-handed or left-handed.

Object Oriented Design:

Object - Chopstick
Operations - Pick_Up, Put_Down

Object - Philosopher
Operations - Give_Names

Object - Console
Operations - Display

```
with Wrap_Around,
procedure Diners is
  type Names is (Socrates, Plato, Buddha, Einstein, Aristotle);

  task type Chopstick is
    entry Pick_Up;
    entry Put_Down;
  end Chopstick;

  task type Philosopher is
    entry Give_Names (My_Name, First_Stick,
                      Second_Stick : in Names);
  end Philosopher;

  -- Each chopstick carries the name of the philosopher to
  -- its right.
  Chopsticks   : array (Names) of Chopstick;
  Philosophers : array (Names) of Philosopher;

  task Console is
    entry Display (Message : in String);
  end Console;

  task body Console     is separate;
  task body Chopstick   is separate;
  task body Philosopher is separate;

  function Wrap is new Wrap_Around (Names);

begin

  -- Tell each philosopher his name.
  Philosophers (Names'First).Give_Names
           ( My_Name      => Names'First,
             First_Stick  => Wrap(Names'First),
             Second_Stick => Names'First );

  for Loop_Index in Wrap (Names'First) .. Names'Last
   loop
      Philosophers (Loop_Index).Give_Names
           ( My_Name      => Loop_Index,
             First_Stick  => Loop_Index,
             Second_Stick => Wrap(Loop_Index) );
   end loop;
 end Diners,
```

```
with Text_IO,
separate (Diners)
task body Chopstick is
 begin
   loop
     select
       accept Pick_Up,      -- Callers will be queued here.
       accept Put_Down;     -- Resource is released here.
     or
       terminate;           -- Server task offers to quit.
     end select;
   end loop;

   exception
     when others =>
       Text_IO.Put_Line ("Chopstick task died.");

  end Chopstick;
```

```
with Text_IO;
separate (Diners)
task body Philosopher is
  My_Name,
  First_Stick,
  Second_Stick : Names;
 begin
   accept Give_Names (My_Name,
                      First_Stick,
                      Second_Stick : in Names) do
     Philosopher.My_Name      := My_Name;
     Philosopher.First_Stick  := First_Stick;
     Philosopher.Second_Stick := Second_Stick;
  end Give_Names;

  declare
    Eating_Message   : constant String :=
             Names'Image(My_Name) & " eating.";

    Thinking_Message : constant String :=
             Names'Image(My_Name) & " thinking.";
  begin

    loop

      Chopsticks    (First_Stick).Pick_Up;
      Chopsticks (Second_Stick).Pick_Up;

      Console.Display (Eating_Message);

      Chopsticks    (First_Stick).Put_Down;
      Chopsticks (Second_Stick).Put_Down;

      Console.Display (Thinking_Message);

    end loop;
  end;

  exception
    when others =>
      Text_IO.Put_Line ("Philosopher task died.");

end Philosopher;
```

```
with Text_IO;
separate (Diners)
task body Console is
  begin
     loop
       select
          accept Display (Message : in String) do
             Text_IO.Put_Line (Message);
          end Display;
       or
          terminate;          -- Server task offers to quit.
       end select;
    end loop;


  exception
    when others =>
       Text_IO.Put_Line ("Console task died.");

  end Console;
```

Ada Generics

by

LCdr Lindy Moran
USNA

Maj Chuck Engle
SEI

Part of the
Advanced Ada Workshop
sponsored by the
Ada Software Engineering Education
& Training Team (ASEET)

Acknowledgements:

Much of this material was taken from earlier Aseet Advanced Generics Tutorials - in particular the tutorial of 18 Aug '87 presented by Mr. John Bailey of Software Metrics, Inc.

GENERICS

- ☐ Why program at all?
- ☐ Why program generically?
- ☐ What does generics provide?
- ☐ How do you write a generic unit?
  - ☐ Parameterless Generics
  - ☐ Parameterized Generics
    - ☐ Value and Object Parameters
    - ☐ Type Parameters
    - ☐ Subprogram Parameters
- ☐ What are the Cons of generics?
- ☐ What are the Pros of generics?
- ☐ What are the unresolved issues?
- ☐ How do you teach generics?

Why program at all?

- Reusability - a programmed solution can be used over and over

- Reliability - program can be tested and verified to ensure correct results for subsequent runs

- Readability - program formalizes human solution and represents it in more abstract readable form

- Maintainability - making a change to a program ensures that the change is consistently applied to all problem solutions

Why program generically?

- Reusability - similar program units needed but different enough to preclude simply entering differing values at run time

- Reliability - generic unit once tested and verified does not need to be retested for each new use or "instantiation"

- Readability - using generic unit allows extraction of the "essence" of the unit eliminating application specific details and produces a very uncluttered readable unit

- Maintainability - a change made to the unit applies to all uses of the unit

Traditional Programming

Algorithms, Objects, Resources

-- intermixed with --

Problem specifics

The Price of Strong Typing
An Example

```
procedure Swap(X,Y : in out INTEGER                    ) is
  Temp : INTEGER       ;
begin
  Temp := X;
  X := Y;
  Y := Temp;
end Swap;

procedure Swap(X,Y : in out CHARACTER                  ) is
  Temp : CHARACTER ;
begin
  Temp := X;
  X := Y;
  Y := Temp;
end Swap;

procedure Swap(X,Y : in out FLOAT                      ) is
  Temp : FLOAT         ;
begin
  Temp := X;
  X := Y;
  Y := Temp;
end Swap;


type GRADE is range 0 .. 100;
procedure Swap(X,Y : in out GRADE                      ) is
  Temp : GRADE         ;
begin
  Temp := X;
  X := Y;
  Y := Temp;
end Swap;
```

Generic Programming

Algorithms, Objects, Resources

---

separated from

Problem specifics

A "generic" Swap Procedure

```
generic
   type ELEMENT is private;
procedure Swap(X,Y : in out ELEMENT);

procedure Swap(X,Y : in out ELEMENT) is
   Temp : ELEMENT;
begin
   Temp :- X;
    X :- Y;
    Y :- Temp;
end Swap;

procedure SwapThings is
   X : integer :- 5;
   Y : integer :- 10;
   Letter1 : character :- 'A';
   Letter2 : character :- 'Z';

   procedure IntSwap is new Swap(integer);
   procedure CharacterSwap is
        new Swap(ELEMENT->character);

begin
   IntSwap(X,Y);
   CharacterSwap(Letter1,Letter2);
end SwapThings;
```

Syntax and Semantics

generic
. . . formal parameters go here . . .
subprogram or package specification

subprogram or package body
. . . body goes here . . .

instantiation to create a usable unit

What does generics provide?

- Generics serve as "templates" for creating or "instantiating" similar conceptual "chunks" of code (packages, functions, or procedures)

- Generics allow removing the problem specifics from a program unit adding greater clarity to its understandability

- Generics allows the programmer to introduce a level of abstraction to increase program understandability

- Generics reduce user's source code size thereby making it more readable and maintainable

- Generics enhance REUSE of software components, facilitating modular system development and easier verifiability

- Generics provide an elegant solution to the restrictions imposed by strong typing

- Generics provides a mechanism for passing subprograms as parameters

- Generics provides a mechanism for doing IO (if needed) for all predefined and user-defined types

Parameterless Generics
"Cloning" Units

A nongeneric "unique object" Stack package:

```
package Stack is
  procedure Pop(I : out integer);
  procedure Push(I : in integer);
  function Empty return boolean;
  function Full return boolean;
end Stack;
```

A non-generic "many objects" solution:

```
package Stacks is
  type Stack is . . .;
  procedure Pop(S : in out Stack; I : out integer);
  procedure Push(S : in out Stack; I : in integer);
  function Empty(S : Stack) return boolean;
  function Full(S : Stack) return boolean;
end Stacks;

-- changes must be made to body of package also
```

A sample user program:

```
procedure StackUp is
  S1, S2 : Stack;   Item : integer;
begin
  Push(S1,10); Push(S2,5); Pop(S1,Item);
end;
```

Parameterless Generics cont.

A generic "many objects" solution:

```
generic
package Stack is
  procedure Pop(I : out integer);
  procedure Push(I : in integer);
  function Empty return boolean;
  function Full return boolean;
end Stack;

-- generic body is identical to non-generic one
-- no changes have to be made to get many stacks
```

A sample user program:

```
with Stack;
procedure StackUp is
   Item : integer;
   package S1 is new Stack;
   package S2 is new Stack;
begin
  S1.Push(10);  S2.Push(5);
  S1.Pop(Item);   S2.Pop(Item);
end StackUp;
```

Parameterless Generics cont.

- Stack implementations compared
  - Non-generic package - only one elaboration and initialization occur
  - Generic package - multiple elaborations and initializations occur - once for each package

Example:
```
with Text_IO;
package body Stack is
  ...
begin
  Text_IO.Put("New stack created.");
end Stack;

package S1 is new Stack; -- message prints
package S2 is new Stack; -- message prints again
pacakge S3 is new Stack; -- message prints again
 . . .
```

Parameterless Generics
"Cloning" Things

"Making The Mold"

```
package VDU is
   subtype Y_Range is integer range 1 .. 24;
   subtype X_Range is integer range 1 .. 80;
   procedure Write(S : in string);
      -- writes S to screen at current cursor loc
   procedure Move(Y : in Y_Range; X : X_Range);
      -- changes cursor position to (X,Y)
   . . .
end VDU;
```

```
generic
package VDU is
   subtype Y_Range is integer range 1 .. 24;
   subtype X_Range is integer range 1 .. 80;
   procedure Write(S : in string);
      -- writes S to screen at current cursor loc
   procedure Move(Y : in Y_Range; X : X_Range);
      -- changes cursor position to (X,Y)
   . . .
end VDU;
```

Generic Instantiation
"Cloning" Things Continued...

"Making The Copies"

```
package VDU1 is new VDU;
package VDU2 is new VDU;
```

```
VDU1.Write("VDU 1");    VDU2.Write("VDU 2");
```

**What if we included "Use VDU1, VDU2;" ?
Would we still need to be explicit and use the package name and dot prefix notation?

[*VDU example taken from ADA Language and Methodology by Watt, Wichmann, and Findlay]

Creating Library Units of Generic Instantiations

```
-- compile following separately into the library

     with Stack;
     package S1 is new Stack;


-- S1 is now a usable library unit

with S1; use S1;
procedure StackUp is
  Item : integer;
begin
  Push(10);
  Push(20);
  Pop(Item);
end StackUp;
```

Parameterized Generics

- Generic Parameters
  - Value and Object Parameters
  - Type Parameters
  - Subprogram Parameters

Value and Object Parameters

- Value Parameters
  - Are of mode IN
  - Serve as local constants in generic units
- Object Parameters
  - Are of mode IN OUT
  - Serve as global objects in generic units

Value Parameters

```
generic
  Max : in integer;
  Min : integer;   -- default mode is IN
procedure BigNSmall(X : integer);

procedure BigNSmall(X : in integer) is
begin
   if X > Max then
     Max := X;          -- not with mode IN
   end if;
   if X < Min then
     Min := X;          -- not with mode IN
   end if;
end BigNSmall;
```

Value Parameters
and
Initialization Before Instantiation

- Actual parameters which are to match with formal generic value parameters must have been initialized before the instantiation occurs

Example:

```
generic
     Max : in integer;
     Min : integer;   -- default mode is IN
procedure BigNSmall(X : integer);

procedure UseBigNSmall is
   LocalMin : integer;         --no initial value
   LocalMax : integer;         -- no initial value
   X : integer :=100;

   procedure Extremes is new
        BigNSmall(Max=>LocalMax,Min=>LocalMin);
   -- error occurs due to lack of initialization

begin
   Extremes(X);
end UseBigNSmall;
```

Value Parameters
and
Levels of Abstraction

```
generic
   Lower, Upper : in character;
function In_Range(S : in string) return boolean;

function In_Range(S : in string) return boolean is
begin
   for I in S'Range loop
     if S(I) not in Lower..Upper then
        return FALSE;
     end if;
   end loop;
   return TRUE;
end In_Range;
```

A non-generic version of In_Range:

```
function In_Range(S : in string; Upper,Lower :
        character) return boolean is
begin
   for I in S'Range loop
      if S(I) not in Lower .. Upper then
         return FALSE;
      end if;
   end loop;
   return TRUE;
end In_Range;
```

Value Parameters and Levels of Abstraction cont.

- Compare clarity in user's programs using generics to add another level of abstraction in "customized" names for In_Range function

```
with In_Range;
procedure InBounds is
  Name : string(1..4) := "JACK";
  Phone : string(1..7) := "6725643";
begin
  if In_Range(Name,'A','Z') then ...
  if In_Range(Phone,'0','9') then ...
end InBounds;
```

```
with In_Range;
procedure InBounds is
  Name : string(1..4) := "JACK";
  Phone : string(1..7) := "6725643";

  function Is_All_Upper_Case is new In_Range('A','Z');

  function Is_All_Lower_Case is new In_Range('a','z');

  function Is_All_Decimal is new In_Range('0','9');

begin
  if Is_All_Upper_Case(Name) then ...
  if Is_All_Decimal(Phone) then ...
end InBounds;
```

[*In_Range taken from Ada Language and Methodology]

Object Parameters

Our Stack Example Revisited

```
generic
  Size : in natural;
pacakge Stacks is
  type Stack is limited private;
  procedure Push(S : in out Stack; I : in integer);
  procedure Pop(S : in out Stack; I : out integer);
private
  subtype NumberOfElements is integer
    range 0..Size;
  type ElementArray is
    array(NumberOfElements) of integer;
  type Stack is record
    Elements : Element_Array;
    Top : NumberOfElements := 0;
  end record;
end Stacks;

with Stacks;
procedure StackUp is
  package SmallStack is new Stacks(5);
  pacakge BigStack is new Stack(5000);
begin
   . . .
end StackUp;
```

Object Parameters and Default Values

```
generic
  Rows : in positive := 24;
  Columns : in positive := 80;
package Terminal is
 . . .
end Terminal;

-- some possible instantiations

package MicroTerminal is new Terminal(24,40);
-- using positional notation

package WordProcessor is new
    Terminal(Columns=>85,Rows=>66);
-- using named notation

package DefaultTerminal is new Terminal;
-- using the default values of 24 and 80
```

Object Parameters and The Subtleties of Default Values

What are the outputs of the following?

```
package CountingPackage is
  function NextNum return integer;

  generic
     Val : integer := NextNum;
  procedure Count;
end CountingPackage;

with Text_IO;
package body CountingPackage is
  CurrentValue : integer := 0;
  function Num return integer is
  begin
    CurrentValue := CurrentValue + 1;
    return CurrentValue;
  end Num;

  procedure Count is
  begin
    Text_IO.Put_Line(integer'image(Val));
  end Count;
end CountingPackage;

with CountingPackage;
procedure StartCounting is
  procedure FirstCount is new CountingPackage.Count;
  procedure CountAgain is new CountingPackage.Count;
begin
  FirstCount;
  CountAgain;
end StartCounting;
```

Object Parameters

A More Useful Example

```
generic
  Control_Block : in out DeviceData;
  Kind : in VDU_Kind := Basic_Kind;
package VDU is
  . . .
end VDU;

with VDU;
procedure ManyVDUs is
  DeviceTable : array(1..N) of DeviceData;

  package VDU1 is new
     VDU(DeviceTable(1),Kind_A);
  package VDU2 is new
     VDU(DeviceTable(2),Kind_B);

begin
  . . .
end ManyVDUs;
```

[*Taken from Ada Language and Methodology]

Object Parameters and Subtleties

- Object parameters passed by reference not by copy-restore method

- Object parameters are "aliases" for their actual parameter counterparts

Example:

```
with Text_IO; use Text_IO;
procedure X is
  Global : integer := 99;
  procedure Z(Param : in out integer) is
  begin
    Param := Param + 1;
    Put_Line(integer'image(Param));
    Put_Line(integer'image(Global));
  end Z;
begin
  Z(Global);
end X;

-- output is 100, 99 for copy-restore method
-- output is 100,100 for pass by reference
```

Object Parameters and Subtleties cont.

- Object parameters passed by reference not by name -- not like Algol's "copy rule"

- Address of actual parameter corresponding to formal generic object parameter is evaluated ONCE and does not change

- Using generic object parameter NOT like doing textual substitution of actual parameter's name

```
declare
   Y : array(1..5) of character := "kitty";
   Index : integer := 1;

   generic
      X : in out character;
   procedure Replace;

   procedure Replace is
   begin
      Index := 5;
      X := 'w';                   -- X => Y(1), NOT Y(5)
      Put(String(Y));
   end Replace;

   procedure Update is new Replace(Y(Index));
   -- Index = 1 when this instantiation occurs

begin
    Update;
end;
```

Object Parameters and Subtleties cont.

- ADDRESS of actual parameter corresponding to a generic formal object parameter is evaluated at time of instantiation -- VALUE in that address not evaluated or copied into the formal parameter

```
declare
  subtype Small is integer range 1 .. 10;
  X : integer := 27;
  generic
    S : in Small;
  procedure Gen;
  procedure Gen is
  begin
     Put("All OK");
  end Gen;
  procedure P is new Gen(X);
  -- Constraint_Error raised at time of instant.
begin
   P;
end;

declare
  subtype Small is integer range 1..10;
  X : integer := 27;
  generic
    S : in out Small;
  procedure Gen;
  procedure Gen is
  begin
     Put("All OK");
  end Gen;
  procedure P is new Gen(X);
  -- executes OK -- would NOT if  value of
  --        S was used inside Gen
begin
  P;
end;
```

Object Parameters

- Use not recommended because suffer from all same falacies as global objects

- Generic object parameters usually SHOULD have been regular formal parameters in the subprogram

Object Parameters cont.

```
generic
  Variable : in out integer;
  Limit, ResetValue : in integer;
procedure ResetIntegerTemplate;

procedure ResetIntegerTemplate is
begin
  if Variable > Limit then
    Variable := ResetValue;
  end if;
end ResetIntegerTemplate;
```

Better written as . . .

```
generic
  Limit, ResetValue : in integer;
procedure ResetIntegerTemplate(Variable : in out
   integer);

procedure ResetIntegerTemplate(Variable : in out
    integer)  is
begin
  if Variable > Limit then
    Variable := ResetValue;
  end if;
end ResetIntegerTemplate;
```

Type Parameters

- type *identifier* is range <>;
- type *identifier* is digits <>;
- type *identifier* is delta <>;
- type *identifier* is (<>);
- type *identifier* is array(*typemark* range <>, . . ., *typemark* range <>) of *typemark*;
- type *identifier* is array(*typemark*, . . . , *typemark*) of *typemark*;
- type *identifier* is access *typemark*;
- type *identifier* is private;
- type *identifier* is limited private;

Integer Type Parameters

- type *identifier* is range <>;

- matches an integer type, predefined or user-defined

- operations defined are those defined for integers such as +,-,/,*,**, rem, mod, negation, abs, >, <, =, /=, <=, >=

- attributes defined are those defined for integers such as 'first, 'last, 'succ, . . .

Integer Type Parameters
An Example

```
generic
  type IntType is range <>;
function Increment(X : IntType) return IntType;

function Increment(X:IntType) return IntType is
begin
  return X+1;
end Increment;

with Increment;
procedure IncrementThings is

  type Age is range 0 .. 130;
  type Temp is range -100 .. 100;

  MyAge : Age  30;
  CurrentTemp : Temp := 80;

  function YearOlder is new Increment(Age);
  function TempUp is new
      Increment(IntType=>Temp);

begin
  MyAge := YearOlder(MyAge);
  CurrentTemp := TempUp(CurrentTemp);
end IncrementThings;
```

Float Type Parameters

- type *identifier* is digits <>;
- matches any floating point type, predefined or user-defined
- operations defined are those available for floating point types such as +, -, /, *, **, negation, abs, >, <, =, /=, <=, >=
- attributes defined are those available for floating point types such as 'small, 'large, 'digits, 'mantisa, 'epsilon, . . .
- useful in providing mathematical routines where user can control the precision used

Float Type Parameters
An Example

```
generic
  type FloatType is digits <>;
function Sqrt(X : FloatType) return FloatType;

function Sqrt(X : FloatType) return FloatType is
begin
  . . .
end Sqrt;

with Sqrt;
procedure Rooting is
    type VeryPrecise is digits 7;
    type Imprecise is digits 3;

    X : VeryPrecise :- 0.1234;
    Y : Imprecise :- 0.12;

    function ExactRoot is new Sqrt(VeryPrecise);
    function RoundRoot is new Sqrt(Imprecise);

begin
    X :- ExactRoot(X);
    Y :- RoundRoot(Y);
end Rooting;
```

Discrete Type Parameters

- type *identifier* is (<>);

- matches any discrete type -- includes integer types and enumeration types (boolean also)

- attributes defined are those available for any discrete/scalar type such as 'first, 'last, 'succ, 'pred, 'image, 'value, 'pos, 'val

- operations defined are those defined for discrete/scalar types such as >, <, =, /=, >=, <=

Discrete Type Parameters
An Example

```
generic
  type Element is (<>);
package Sets is
  type Set is private;
  function Intersection(S1,S2 : Set) return Set;
  function Union(S1,S2 : Set) return Set;
  function IsIn(Item : Element; S : Set) return
      boolean;
  function IsNull(S : Set) return boolean;
private
  type Set is array(Element) of boolean;
end Sets;


-- some possible instantiations

package CharacterSet is new Sets(character);

package IntegerSet is new Sets(integer);

type Student is (John, Joan, Ann, Sue, . . ., Zip);
package StudentSet is new Sets(Student);
```

Discrete Type Parameters cont.

☐ Minimal assumptions about the type must be made - operations must apply to ALL discrete types

Example:

```
generic
  type Element is (<>);
function Next(X : Element) return Element;

function Next(X : Element) return Element is
begin
   X := X + 1;     -- not defined for ALL
                   --   discrete types
end Next;
```

Use attributes:

```
function Next(X : Element) return Element is
begin
   if X = Element'Last then
     return Element'First;
   else
     return Element'Succ(X);
   end if;
end Next;
```

Constrained Array Type Parameters

- type *identifier* is array (*typemark*, . . ., *typemark*) of *typemark*;

- matches any constrained array type where:
  1) number of dimensions match,
  2) index subtypes of corresponding dimensions match,
  3) bounds in corresponding dimensions are identical,
  4) component types match

- attributes defined are those available for constrained arrays such as 'first(n), 'last(n), 'range(n), 'length(n)

- operations defined include those available for constrained arrays such as =, :=, using slice notation (for one dimensional arrays)

Constrained Array Type Parameters
An Example

```
generic
   type Component is (<>);
   type AnArray is array(1..10) of Component;
procedure Sort(A : in out AnArray);
procedure Sort(A : in out AnArray) is
  Temp : Component;
begin
  for I in 2 .. 10 loop
    for J in 1..I-1 loop
      if A(I) < A(J) then
        Temp := A(J);
        A(J) := A(I);
        A(I) := Temp;
      end if;
    end loop;
  end loop;
end Sort;

-- in user program
type Age is integer range 0..130;
type AgeArray is array(1..10) of Age;
X : AgeArray := (8,0,9,4,50,35,87,97,1,124);

  procedure AgeSort is new
      Sort(Component,AgeArray);

...  AgeSort(X); ...
```

Unconstrained Array Type Parameters

- type *identifier* is array(*typemark* range <>, . . ., *typemark* range <>) of *typemark*;

- matches any unconstrained array where:
  1) number of dimensions the same
  2) subtype of index for corresponding dimensions is the same
  3) component types match

- attributes defined are those available for unconstrained arrays such as 'first(n), 'last(n), 'range(n), 'length(n)

- operations defined include those available for unconstrained arrays such as =, :=, using slice notation (for one dimensional typearrays)

Unconstrained Array Type Parameters
An Example

```
generic
  type Index is range <>;
  type Component is range <>;
  type AnArray is array(Index) of Component;
procedure Sort(A : in out AnArray);
procedure Sort(A : in out AnArray) is
  Temp : Component;
begin
  for I in A'First+1 .. A'Last loop
    for J in A'First .. I-1 loop
      if A(I) < A(J) then
        Temp := A(J);
        A(J) := A(I);
        A(I) := Temp;
      end if;
    end loop;
  end loop;
end Sort;

--in user's program
type Age is range 0..130;
type EmployeeNumber is range 1..100;
type EmpList is array(EmployeeNumber) of Age;
  procedure EmployeeAgeSort is new
    Sort(EmployeeNumber,Age,EmpList);
    Employees : EmpList := (. . . .);

. . . EmployeeAgeSort(Employees); . . .
```

Private Type Parameters

- type *identifier* is private;
- matches any type except a limited type
- operations available are only declaring objects of the type, testing for equality and inequality, and assigning values to objects of the type

Private Type Parameters
An Example

```
generic
  type Index is (<>);
  type Component is private;
  type AnArray is array(Index) of Component;
function Found(A : AnArray; T : Component)
    return boolean;
function Found(A : AnArray; T : Component)
   return boolean is
begin
  for I in A'First..A'Last loop
    if A(I) = T then
      return TRUE;
    end if;
  end loop;
  return FALSE;
end Found;

--in user's program
type Student is (Joan,John,Sue,...,Debbie);
type Grade is range 0..100;
type GradeArray is array(Student) of Grade;
function GradeMade is new
    Found(Student,Grade,GradeArray);
Grades : GradeArray := (. . . .);

. . .  if GradeMade(Grades,100) then . . .
```

Private Type Parameters cont. and Restrictions Imposed

What's wrong here?

```
generic
   type Index is (<>);
   type Component is private;
   type Int_Array is array(Index) of Component;
procedure Sort_Array(Arr : in out Int_Array);

procedure Sort_Array(Arr : in out Int_Array) is
 Temp : Component;
begin
  for I in Index'Succ(Arr'First)..Arr'Last loop
   for J in Arr'First..Index'Pred(I) loop
     if Arr(I) < Ar(J) then
       Temp := Arr(J);
       Arr(J) := Arr(I);
       Arr(I) := Temp;
     end if;
   end loop;
  end loop;
end Sort_Array;
```

Private Type Parameters
Another Caution

What's wrong here?

```
generic
   type Element is private;
procedure Swap(X,Y : in out Element);

procedure Swap(X,Y : in out Element) is
    Temp : Element;
begin
   Temp := X;
   X := Y;
   Y := Temp;
end Swap;

-- in user's program
HerName : string(1..5) := "Lindy";
HisName : string(1..5) := "Chuck";

procedure NameSwap is new Swap(string);
```

Limited Private Type Parameters

- matches any type including a limited type

- only declaration of objects of the type permitted and NOTHING else

Example:

```
generic
   MyFile : Text_IO.File_Type; -- illegal
procedure Oops;
```

Access Type Parameters

- matches any access type

- operations defined for access types available such as setting object to null, use of NEW allocator, use of .ALL notation

-- Example follows introduction of subprogram parameters

Generic Formal Type Parameters
A Synopsis

Generic formal parameter	Actual parameter
type T is limited private;	any type
type T is private;	any non-limited type
type T is (<>);	any discrete type
type T is range<>;	any integer type
type T is digits <>;	any float type
type T is delta <>;	any fixed point type

[*Taken from Ada Language and Methodology]

Type Parameters and The Standard Generic IO Packages

```
package Text_IO is
   . . . non- generic part of Text_IO
   generic
     type NUM is range <>;
   package Integer_IO is
     . . .
   end Integer_IO;

   generic
     type NUM is digits <>;
   pacakge Float_IO is
     . . .
   end Float_IO;

   generic
     type NUM is delta <>;
   package Fixed_IO is
     . . .
   end Fixed_IO;

   generic
     type ENUM is (<>);
   package Enumeration_IO is
     . . .
   end Enumeration_IO;
end Text_IO;
```

Generic Formal Type Parameters
How To Choose?

- ☐ What operations are performed on the type in the generic body?

- ☐ How restrictive on the type that the user can choose do you want to be?

Subprogram Parameters

- ☐ allow definition and "pass in" of additional operations for generic formal type parameters - especially private and limited private types
- ☐ can pass functions or procedures
- ☐ formal parameters of generic formal subprogram parameter are checked to ensure match with actual parameters in a call to that subprogram at compile time

Subprogram Parameters
and
A Pascal Flaw Resolved

```
program P;
      type Color - (Red,Green,Blue);
      var Bucket : Color;

      procedure Print(C : Color);
      begin
        case C of
          Red : writeln('Red');
          Green : writeln('Green');
          Blue : writeln('Blue');
        end case;
      end;

      procedure Proc(P : procedure);
      begin
        P(Bucket);          (* OK *)
        P(5);               (* RUNtime error *)
      end;

begin
      Proc(Print);
end.
```

Subprogram Parameters
A Pascal Flaw Resolved cont.

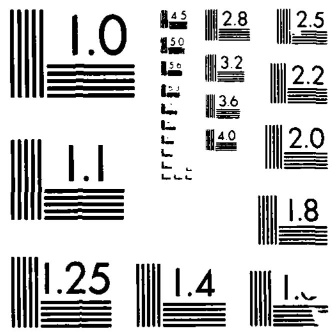
```
declare
  type Color is (Red,Green,Blue);
  Bucket : Color := Green;

  procedure Print(C : in Color) is
  begin
    Text_IO.Put(Color'Image(C));
  end Pring;

  generic
    with procedure P(Val : Color);
  procedure Gen_Proc;

  procedure Gen_Proc is
  begin
    P(Bucket);     -- OK
    P(5);          -- COMPILE time error
  end Gen_Proc;

begin
  Proc;
end;
```

Subprogram Parameters
An Example

```
generic
  type Index is (<>);
  type Component is private;
  type Int_Array is array(Index) of Component;
  with function "<"(X,Y:Component)
     return boolean;
procedure Sort_Array(Arr : in out Int_Array);

procedure Sort_Array(Arr : in out Int_Array) is
  Temp : Component;
begin
  for I in Index'Succ(Arr'First)..Arr'Last loop
  for J in Arr'First..Index'Pred(I) loop
    if Arr(I) < Ar(J) then
      Temp := Arr(J);
       Arr(J) := Arr(I);
      Arr(I) := Temp;
    end if;
  end loop;
  end loop;
end Sort_Array;
```

AD-A192 074 ASEET ADVANCED ADA WORKSHOP JANUARY 1988(U) ADA JOINT PROGRAM OFFICE ARLINGTON VA JAN 88 5/5

UNCLASSIFIED F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Subprogram Parameters
An Example cont.

```
type Day is range 1..31;
type WeatherRec is record
  RainFall : natural;
  AvgTemp : float;
end record;
type WeatherArray is array(Day) of WeatherRec;

function LT(X,Y: WeatherRec) return boolean is
begin
  return X.Rainfall > Y.Rainfall;
end LT;

function "<"(X,Y : WeatherRec) return boolean is
begin
  return X.AvgTemp > Y.AvgTemp;
end "<";

procedure RainSort is new Sort_Array(Day,
   WeatherRec, WeatherArray, LT);

procedure TempSort is new Sort_Array
  (Index=>Day, Component=>WeatherRec,
   Int_Array=>WeatherArray,"<"=>"<");

WeatherData : WeatherArray := ( . . . .);
begin
  RainSort(WeatherData);
  TempSort(WeatherData);  . . . end;
```

Subprogram Parameters and Default Values

```
generic
  type Index is (<>);
  type Component is private;
  type Int_Array is array(Index) of Component;
  with function "<"(X,Y:Component)
      return boolean is <>;
procedure Sort_Array(Arr : in out Int_Array);


--in user's program

function "<"(X,Y : WeatherRec) return boolean is
begin
  return X.AvgTemp > Y.AvgTemp;
end "<";

procedure DefaultSort is new Sort_Array
  (Index=>Day,Component=>WeatherRec,
   Int_Array=>WeatherArray);


...  DefaultSort(WeatherData); -- will sort on
                              -- temp values
```

Subprogram Parameters and Default Values cont.

Another example:

```
type SmallRange is range 1..10;
type Values is array(SmallRange) of integer;

procedure IntegerSort is new Sort_Array
  (Index=>SmallRange, Component=>integer,
   Int_Array=>Values);
-- default > for

  V : Values := (. . . .);
begin
  IntegerSort(V);  -- default "<" for integers used
end;

-- using Put for subprogram parameter name
--    results in default to generic Put routines
--    in the IO packages
```

Subprogram Parameters
and
Subtleties of Default Values

- ☐ Global references inside a generic are resolved to those at point of DECLARATION.

- ☐ For subprogram parameters, default references resolve to matching names from point of INSTANTIATION.

```
with Text_IO; use Text_IO;
package Shell is
  Global : integer := 17;
  generic
    with procedure Put(Val : integer) is <>;
  procedure Demo;
end Shell;

package body Shell is
  procedure Demo is
  begin
    Put(Global);
  end Demo;
end Shell;
------------------------------
with Shell;
package Inner is
  Global : integer := 39;
  procedure Put(I : integer);

  procedure User is new Shell.Demo;
end Inner;

with Text_IO;
package body Inner is
  procedure Put(I : integer) is
  begin
    Text_IO.Put("Surprise" & integer'image(I));
  end Put;
end Inner;

... Inner.User; ...
```

Subprogram Parameters
and
Access Type Parameters
An Example

```
generic
   type KeyType is private;
   type ElementType is private;
   with function "<"(Left,Right : KeyType)
      return boolean is <>;
package BinaryTreeMaker is
   type Kind is private;
   function Make return Kind;
   function IsEmpty(T : Kind) return boolean;
   procedure Insert(T : in out Kind;
                    K : KeyType;
                    E : ElementType);
   function Retrieve(T : Kind; K : KeyType)
      return ElementType;
   KeyNotFound : exception;

   generic
      with procedure Operation(K : KeyType;
         E : ElementType);
   procedure InorderTraverse(TheTree: in Kind);
private
   type InternalRecord;
   type Kind is access InternalRecord;
end BinaryTreeMaker;
```

```
with BinaryTreeMaker;
package EmployeeDataBase is
   NameLength : constant := 40;
   subtype NameType is string(1..NameLength);
   type Dollar is delta 0.01 range 0.0..1.0e8;
   type AgeType is range 0 .. 150;
   type YearType is range 1900..2100;
   type EmployeeInfo is record
      Salary : Dollar;
      Age : AgeType;
      Hired : YearType;
   end record;

   package EmployeeTree is new
      BinaryTreeMaker(KeyType=>NameType,
                      ElementType=>EmployeeInfo);

   RootNode : EmployeeTree.Kind;
end EmployeeDataBase;
```

```
with EmployeeDataBase; use EmployeeDataBase;
with Text_IO; use Text_IO;
procedure PrintReports is
   package SalaryIO is new Fixed_IO(Dollar);
   package AgeIO is new Integer_IO(AgeType);
   use SalaryIO, AgeIO;

   procedure PrintSalary(Key : NameType;
      Info : EmployeeInfo) is
   begin
      . . . Put(Info.Salary);
   end;

   procedure Print Age(Key : NameType;
      Info : EmployeeInfo) is
   begin
      . . . Put(Info.Age);
   end;

   procedure ReportSalaries is new
      EmployeeTree.InorderTraverse
         (Operation-> PrintSalary);

   procedure ReportAge is new
      EmployeeTree.InorderTraverse
         (Operation-> PrintAge);
begin
   ReportSalaries(RootNode);
   New_Line;
   ReportAges(RootNode);
end PrintReports;
```

[*From Understanding Ada]

Subprogram Parameters and Handling Exceptions

```
generic
package Stack is
  . . . same as before

  Overflow, Underflow : exception;
end Stack;

-- in user's program

  package S1 is new Stack;
  package S2 is new Stack;

begin
  S1.Push(5);
  S2.Pop(Item);
exception
  when S1.Underflow => . . .;
  when S1.Overflow => . . .;
  when S2.Underflow => . . .;
  when S2.Overflow => . . .;
end;
```

Subprogram Parameters and Handling Exceptions cont.

- Cannot pass exceptions as generic parameter

```
generic
  When_Error : exception;   -- NOT allowed
   . . .
procedure X . . .
  . . .
exception
  when others => raise When_Error;
end X;


My_Exception : exception;
procedure S is new X(My_Exception);

. . .
begin
    S;
exception
  when My_Exception =>  . . .;  -- NOT allowed
end;
```

Subprogram Parameters and Handling Exceptions cont.

```
generic
  with procedure OverflowHandler;
package Stack is
  . . . same as before;
end Stack;

package body Stack is

. . . in Push procedure . . .
  when Constraint_Error => OverflowHandler;

end Stack;

-- in user program
with Stack;
. . .
procedure OverflowHandler is
begin
  Text_IO.Put_Line("Overflow has occurred");
end OverflowHandler;

package S1 is new Stack(OverflowHandler);

begin
  . . .
  S1.Push(5);  -- if overflow occurs msg prints
end;
```

Generic Can'ts

- No generic SUBtype parameters, only TYPEs
- No generic record types
- No generic tasks
  - Wrap a package around it

What are the Cons of Generics?

- ☐ Takes longer/is harder to write generic code
- ☐ Usually some efficiency sacrificed for the generality -- use of application specifics could lead to increased efficiency
- ☐ Difficult to make component robust/reliable enough to survive all uses

What are the Pros of generics?

- Reusability - no reinventing the wheel for each specific application

- Levels of abstraction added - separation of abstraction and implementation

- Source code size of user programs reduced
  - Maintainability, readability, and understandability increased
  - Verification more manageable

- When used in conjunction with user-defined types increases portability across machines

- Provides necessary answer to strong typing without sacrificing increased reliability of compile time checks

- Provides flexible IO packages which can be used (if needed) for predefined AND user-defined types

Unresolved Issues in Generics

- Compiler Issues
  - Use "code sharing" or "code copying" to implement generics
- Management Issues
  - How to facilitate creation of generic units
    - In retrospect, after recognizing similarity in produced units
    - Beforehand using "domain analysis"
  - How to manage storage and retrieval of units in a library of generic units
  - How to "publicize" availability of units in generic library and provide criterion for selecting proper unit
  - How to manage updating of used generic units as "bugs" are uncovered
- Legal Issues
  - Who owns the generic module
  - Who is liable for the generic module's performance

[*See Software Components with Ada]

How do you TEACH generics?

- Necessary as IO is an issue arising early and should not be kept a "magic" process
- One key is to use concrete examples
  - Driver's licence form is a generic template -- individual's license is a usable instantiation
- One key is to tie to previous learning
  - Use old/familiar packages, procedures, and functions - Stacks, Swap, etc.

END
DATE
FILMED
6-88
DTIC